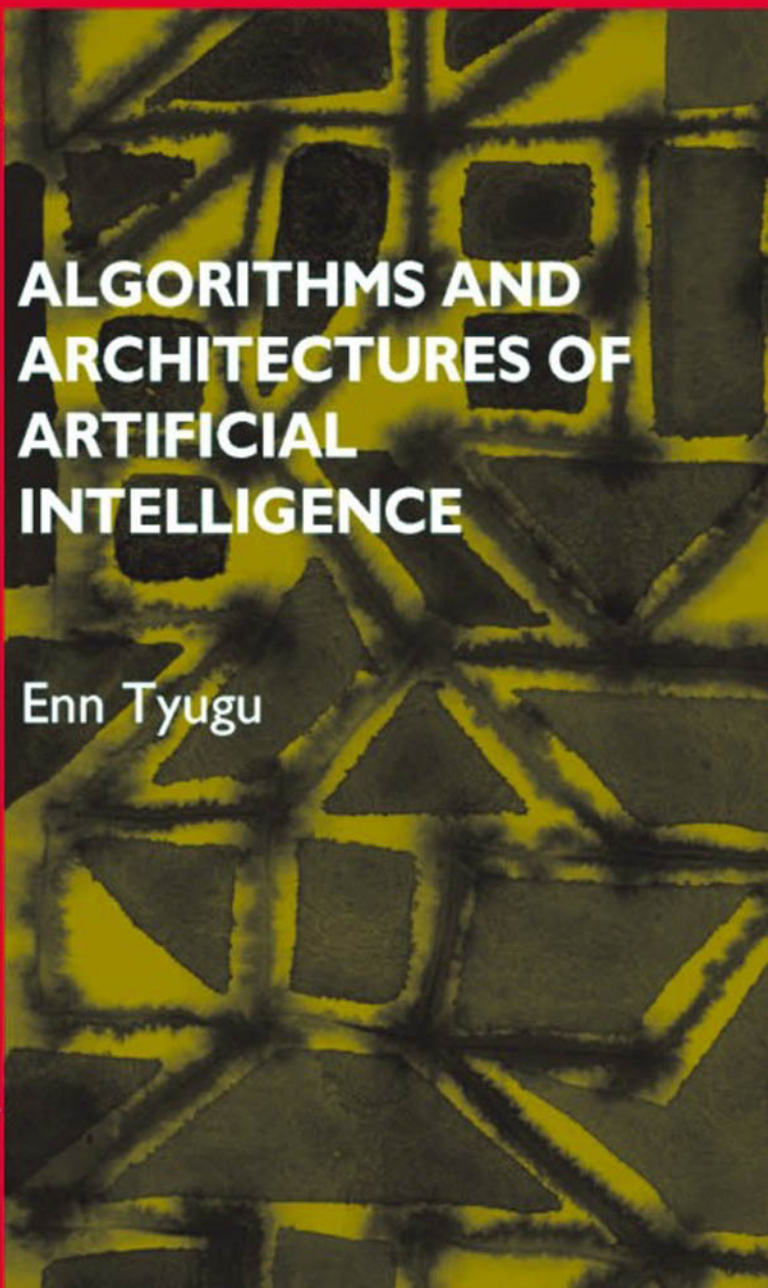
Frontiers in Artificial Intelligence and Applications



# ALGORITHMS AND ARCHITECTURES OF ARTIFICIAL INTELLIGENCE

Enn Tyugu

IOS Press

# ALGORITHMS AND ARCHITECTURES OF ARTIFICIAL INTELLIGENCE

# Frontiers in Artificial Intelligence and Applications

FAIA covers all aspects of theoretical and applied artificial intelligence research in the form of monographs, doctoral dissertations, textbooks, handbooks and proceedings volumes. The FAIA series contains several sub-series, including “Information Modelling and Knowledge Bases” and “Knowledge-Based Intelligent Engineering Systems”. It also includes the biennial ECAI, the European Conference on Artificial Intelligence, proceedings volumes, and other ECCAI – the European Coordinating Committee on Artificial Intelligence – sponsored publications. An editorial panel of internationally well-known scholars is appointed to provide a high quality selection.



## Volume 159

*Recently published in this series*

Vol. 158. R. Luckin et al. (Eds.), Artificial Intelligence in Education – Building Technology Rich Learning Contexts That Work
Vol. 157. B. Goertzel and P. Wang (Eds.), Advances in Artificial General Intelligence: Concepts, Architectures and Algorithms – Proceedings of the AGI Workshop 2006
Vol. 156. R.M. Colomb, Ontology and the Semantic Web
Vol. 155. O. Vasilecas et al. (Eds.), Databases and Information Systems IV – Selected Papers from the Seventh International Baltic Conference DB&IS’2006
Vol. 154. M. Duží et al. (Eds.), Information Modelling and Knowledge Bases XVIII
Vol. 153. Y. Vogiazou, Design for Emergence – Collaborative Social Play with Online and Location-Based Media
Vol. 152. T.M. van Engers (Ed.), Legal Knowledge and Information Systems – JURIX 2006: The Nineteenth Annual Conference
Vol. 151. R. Mizoguchi et al. (Eds.), Learning by Effective Utilization of Technologies: Facilitating Intercultural Understanding
Vol. 150. B. Bennett and C. Fellbaum (Eds.), Formal Ontology in Information Systems – Proceedings of the Fourth International Conference (FOIS 2006)
Vol. 149. X.F. Zha and R.J. Howlett (Eds.), Integrated Intelligent Systems for Engineering Design
Vol. 148. K. Kersting, An Inductive Logic Programming Approach to Statistical Relational Learning
Vol. 147. H. Fujita and M. Mejri (Eds.), New Trends in Software Methodologies, Tools and Techniques – Proceedings of the fifth SoMeT_06

ISSN 0922-6389

# Algorithms and Architectures of Artificial Intelligence

Enn Tyugu
*Institute of Cybernetics, Tallinn University of Technology, Estonia*

IOS Press
Amsterdam • Berlin • Oxford • Tokyo • Washington, DC

ISBN 978-1-58603-770-3
Library of Congress Control Number: 2007930585

*Publisher*
IOS Press
Nieuwe Hemweg 6B
1013 BG Amsterdam
Netherlands
fax: +31 20 687 0019
e-mail: order@iospress.nl

*Distributor in the UK and Ireland*
Gazelle Books Services Ltd.
White Cross Mills
Hightown
Lancaster LA1 4XS
United Kingdom
fax: +44 1524 63232
e-mail: sales@gazellebooks.co.uk

*Distributor in the USA and Canada*
IOS Press, Inc.
4502 Rachael Manor Drive
Fairfax, VA 22032
USA
fax: +1 703 323 3668
e-mail: iosbooks@iospress.com



PRINTED IN THE NETHERLANDS

*Algorithms and Architectures of Artificial Intelligence*
*E. Tyugu*
*IOS Press, 2007*



# Foreword

This book gives an overview of methods developed in artificial intelligence for search, learning, problem solving and decision-making. It gives an overview of algorithms and architectures of artificial intelligence that have reached the degree of maturity when a method can be presented as an algorithm, or when a well-defined architecture is known, e.g. in neural nets and intelligent agents. It can be used as a handbook for a wide audience of application developers who are interested in using artificial intelligence methods in their software products. Parts of the text are rather independent, so that one can look into index and go directly to a description of a method presented in the form of an abstract algorithm or an architectural solution. The book can be used also as a textbook for a course in applied artificial intelligence. It has been used as a course material for this kind of courses on undergraduate and graduate levels at Tallinn University of Technology and Royal Institute of Technology in Stockholm. Exercises on the subject are added at the end of each chapter. Neither programming skills nor specific knowledge in computer science are expected from the reader. However, some parts of the text will be fully understood by those who know the terminology of computing well.

The algorithms developed in the field of artificial intelligence are often directly applicable in solving various difficult problems and, therefore, they constitute a valuable part of the artificial intelligence that can be used independently of the philosophical and deep theoretical considerations related to the research in this area. Most of the algorithms are presented in this book in a simple algorithmic language. Sometimes, e.g. when describing neurons in artificial neural nets, it has been more expressive to use common mathematical notations for presenting the functionality of entities. Taxonomy of algorithms and methods concerned is discussed at the end of each chapter, summarizing the material presented in it. This taxonomy can be a valuable aid in developing new algorithms by elaborating the presented ones.

This page intentionally left blank

# Contents

This page intentionally left blank

*Algorithms and Architectures of Artificial Intelligence*
*E. Tyugu*
*IOS Press, 2007*



# Introduction

There are two different understandings of artificial intelligence (AI) as a research field. One approach takes a philosophical attitude, and is interested in the possibilities of building artificial systems with intelligent behavior primarily for scientific purposes – to understand intelligence and to test the abilities of computers. This attitude prevailed in the beginning of the AI research starting from the middle of the last century. Another attitude is aimed at practical applications, and is interested in building applications that possess some intelligence, or more precisely, seem to be intelligent. This approach became very popular with the development of expert systems about thirty years ago. It continued with the research and applications of artificial neural nets, constraint solving and intelligent agents. It includes even more specific areas like robotics, natural language understanding etc. Many useful and quite intelligent methods of data processing have been developed in AI during the years. A number of them have become widely used, like various search methods, planning, constraint solving and optimization methods.

This book is related to the practical approach in artificial intelligence. It is a collection of useful methods and architectural solutions developed during the years of the AI research and application. Some of these methods are well known and are even not considered belonging particularly to AI any more. Almost all methods included in this book are tested in practice and are presented either in the form of an algorithm or are included into an architecture of a software system like, for instance, a system of cooperating intelligent agents or a neural net composed of artificial neurons.

We start the presentation with a brief introduction to a language of algorithms that used further in all four chapters of the book.

It is obvious that intelligence is closely related to knowledge and ability of knowledge handling. One can say that knowledge lies in the basis of artificial intelligence as well. More generally – knowledge has the same role in artificial intelligence (and in information technology in general) that energy has in other fields of technology and engineering – the more one has it the more can be achieved. The book begins with a chapter on knowledge representation and knowledge handling. An introduction into knowledge systems is given, where basic knowledge-handling and inference algorithms are presented, starting with simple brute force deduction and value propagation algorithms, and continuing with more elaborate methods. Resolution method and different resolution strategies are discussed in more detail. Other knowledge representation methods discussed in the first chapter are semantic networks, frames, production rules and decision trees. Also architecture of knowledge-based systems is discussed in the first chapter, and a precise notation for describing knowledge architecture of intelligent systems is presented.

Search algorithms are a classical and well-developed part of the artificial intelligence. They are the basis of learning and problem solving algorithms. Search is a universal method of problem solving that can be applied in all cases when no other methods of problem solving are applicable. This makes search algorithms the most valuable and frequently used part of artificial intelligence algorithms. In Chapter 2, various exhaustive search and heuristic search methods are systematically explained and represented in the form of algorithms. This chapter includes about thirty search algorithms that can be used for solving different problems.

Learning and decision making is considered in Chapter 3. This chapter begins with simple algorithms of parametric and adaptive learning as well as learning in automata. It continues with symbolic learning, including specific to general and general to specific concept-learning methods. This is followed by a discussion of inductive inference and

inductive logic programming. Massively parallel learning methods are represented by genetic algorithms. This chapter includes also descriptions of well-known neural nets as the architecture that supports learning and decision making. A number of specific leaning methods like learning in monotonous systems are explained in the end of this chapter.

Problem solving and planning is a rapidly developing branch of the artificial intelligence that has already given a considerable output into the computing practice. Algorithms of solving constraints on finite domains and constraint propagation methods are considered in Chapter 4. Separate sections are devoted to planning and automatic synthesis of algorithms. One practical program synthesis technique – structural synthesis of programs that is based on sound logical method is presented in more detail. Architectures of intelligent agents are presented where problem solving and planning are applied.

Finally, there is a clarification about the algorithms presented in this book. The algorithms are not software products, and not even pieces of software. The algorithms should be used analogically to formulas from a handbook of numeric methods in applied mathematics. One has to understand an algorithm and adjust it by developing concrete data representation and functions that use the data. In this sense this book is most suitable for system engineers who decide about the methods and design of software.

# Language of algorithms

We use an informal language for presenting algorithms in the following text, preferring a functional style of writing to the imperative style and presenting notations and descriptions of program entities in informal prefaces to algorithms. However, this language is still more formal than an ordinary "formal" language of mathematics used in scientific literature. This allows us to express the ideas of algorithms briefly and, we hope, also comprehensively. We use mathematical notations in expressions of the algorithms rather freely. Sometimes this hides real computational complexities, but enables us to concentrate on the main ideas of algorithms. For the readability, we try to avoid the usage of curly brackets *{* and *}* for blocks, instead, we end loops with the keyword ***od*** and conditional statements with the keyword ***fi***. Semicolon is used for connecting statements in a sequence of statements and not as the end of a statement. Assignment is denoted by = and Boolean equality operator is ==. The following is a summary of notations used in the algorithms.

1. Loops are built with ***while*** and ***for*** control parts and with a body between ***do*** and ***od***.
   Examples***: while not*** *good(x)* ***do*** *search(x)* ***od***
   ***for*** *i=1* ***to*** *n* ***do*** *print*($a_i$) ***od***

2. Set notations are used in loop control and other expressions.
   Example: ***for*** $x \in S$ ***do*** *improve(x)* ***od***

3. Conditional statements are with ***if*** and ***then*** parts, and may contain ***else if*** and ***else*** parts.
   Example: ***if*** $x < y$ ***then*** $m=x$ ***else*** $m=y$ ***fi***

4. Recursive definitions can be used as follows:

Example: *gcd(x,y)* =

```
if x==y then
    return x
else if x>y then
    return gcd(x-y,y)
else
    return gcd(y,x)
fi
```

5. Common functions for list manipulation (*head, tail, append* etc.) are used without introduction.

6. Some predicates and functions are being used without introducing them in the preface. They have the following predefined meaning:

*good(x)* - *x* is a good solution satisfying the conditions given for the problem

*empty(x)* - *x* is an empty object (tree, set, list etc.)

*success()* - actions taken in the case of successful termination of the algorithm, it can be supplied with an argument that will be the output of the algorithm: *success(x)*

*failure()* - actions taken in the case of unsuccessful termination of the algorithm

*break L* - exit a statement labeled *L*

*return x* – ends the algorithm and returns a value of the expression *x*.

In some special cases, when describing learning automata, we have presented algorithms in the form of Java classes. This is justified by the suitability of object-oriented representation of automata that react to input messages. These classes are simple enough to understand them without being an expert in Java.

# 1. Knowledge Handling

## 1.1. Knowledge and knowledge systems

It is well accepted that knowledge is something that can be processed (stored, changed and used) by computers. However, there is no generally accepted definition of knowledge. And what does one mean by using knowledge? Trying to define knowledge seems to be as difficult as to define energy, for instance. One can predict that knowledge will play the same role in computer science that energy has in physics, the more you have it – the more you can achieve. One can say that *knowledge* is the content of data for a user who understands the data. However, this is not a definition even in the weakest sense, because we lack a definition of understanding.

Without having any good definition of knowledge, we still are going to define a knowledge system, and we do it operationally by showing what the system can do. Quite informally, a *knowledge system* is a language for representing knowledge plus a mechanism (a program, for example) for using the knowledge by making inferences. A knowledge system can be implemented in the form of a *knowledge base* and *inference engine*. This is precisely what we have in expert systems, where knowledge is being used for making expert level decisions.

We discuss a number of knowledge systems and present algorithms that are used for knowledge handling in this chapter. We start with abstract and rather formal knowledge representation that can to some extent compensate the absence of a precise definition of knowledge itself. Thereafter we discuss a number of popular knowledge representation forms like Horn clauses, semantic networks, frames, rule-based systems, decision tables etc. We present algorithms of knowledge handling for most of these systems, including a Prolog interpreter. At the end of the chapter we propose a way of expressing knowledge architecture as a composition of knowledge modules that can use different knowledge representation even in one and the same knowledge-based system.

### 1.1.1. Abstract representation of knowledge systems

We define a knowledge system formally in a way that enables us to use this concept for any system that can store and use knowledge for performing some tasks, e.g. obtaining answers to some given questions. Besides knowledge systems, we are going to use a concept of a knowledge-based system. We consider a *knowledge system* as a knowledge-processing module, and a *knowledge-based system* as a composition of modules of this kind. The aim of this section is to outline the concept of knowledge system in a way that is independent of any particular implementation, and is usable in the analysis and explanation of architectures of knowledge-based applications.

First, we assume that knowledge is always represented either by one or several *knowledge objects* which can be, for instance formulae, or just texts in a suitable language, or by a *state* of some system. At a closer look, one can detect that the second form of knowledge representation is reducible (can be transformed) to the representation in the form of knowledge objects.

Second, we assume that to use knowledge means to perform some operations on knowledge objects in such a way that new knowledge objects appear. We do not see other ways of knowledge usage. In the case of knowledge representation as a state of a system, knowledge usage means transforming the state. Also this can be represented as changing the set of knowledge objects that encode the state. These two assumptions have lead us to a

concept of deductive system. For detailed discussion of this formalism we refer to Maslov [29].

Third, we assume that knowledge objects have some meaning that may be objects or maybe even some effects on the environment where they exist. To make it precise, one has to consider a collection of possible meanings and a mapping from knowledge object to meanings. This can be formalized as an *interpretation* of knowledge objects. One can, instead of the mapping, consider a relation that binds knowledge objects with their meanings. This is a *relation of notation-denotation* that is intuitively well known in philosophy and logic.

The conventional concept of a *deductive system* is defined as a set of initial objects and derivation rules for generating new objects from given objects. (See Maslov [29].) We are going to use a slightly different concept – a free deductive system, where initial objects are not fixed. Giving some initial objects in the language of a free deductive system makes it a conventional deductive system.

**Definition.** *Free deductive system* is a language of objects and rules for generating new objects from given objects.

A good example of a free deductive system is a Post's system without initial objects. Post's systems are in knowledge processing like Markov's normal algorithms in computing. The latter are, due to their syntactic simplicity, useful in theory of algorithms, but useless in programming. Post's systems are due to their simplicity, useful in understanding and proving principles of knowledge processing, but almost useless in practical applications. We are going to say more about them in the next section.

**Definition.** *Interpretation* of a (free) deductive system is a set $M$ of entities that are possible meanings of objects of the deductive system, and a relation $\mathsf{R}$ that binds at least one meaning with every object, and binds at least one object (a notation) with every meaning included in $M$. The relation $\mathsf{R}$ is called a *relation of notation-denotation*.

**Remark.** A widely used method to get a meaning of a new object generated by a rule is to construct it from meanings of other objects used in the rule. From the other side, we do not know any general way to find objects that are bound to a meaning by the relation of notation-denotation.

**Definition.** *Knowledge system* is an interpreted free deductive system.

Examples of free deductive systems are natural languages with one derivation rule that generates a new sequence of sentences $ST$ for any correct sequences of sentences $S$ and $T$ of a language. This rule can be written as follows:

$$\frac{S \quad T}{ST}$$

If we are able to get a meaning for any sequence of sentences, and can find a sentence denoting a given meaning, we can say that we have a knowledge system. In other words – we a have a knowledge system, if and only if we understand the language.

Another example of a knowledge system is a *calculus of computable functions* – CCF. This can be any representation of computable functions together with suitable computation rules, e.g. lambda calculus, Turing machine, a programming language etc. In this paper we shall use CCF each time, when we have to represent computations as the principal way of knowledge handling, ignoring the actual computational device, i.e. the implementation.

Sometimes we are interested only in knowledge representation, and not in making inferences, i.e. we need only a language of objects with interpretation. Formally we have then a knowledge system with empty set of derivation rules. We call a knowledge system with empty set of rules a *closed knowledge system.*

Sometimes we are unable (or are just not interested) to find a notation for an arbitrary given meaning, but still can find a meaning for any notation. Then we can use a mapping from notations to denotations instead of the relation of notation-denotation (that binds also a notation with every denotation), and have a weaker notion of knowledge system defined as follows.

*Weak interpretation* of a deductive system is a set of entities that are possible meanings of objects of the deductive system, and a mapping that binds at least one meaning with every object of the deductive system.

*Weak knowledge system* is a free deductive system with weak interpretation. We call a knowledge system *normal*, if we wish to stress that it is not weak.

## 1.1.2. Examples of deductive systems

**Calculus of computability**. Let us have a set of symbols that we call an alphabet. The objects of our language will be formulas of the form: $A \rightarrow B$, where $A$ and $B$ are finite sets of symbols from the alphabet. We can allow also formulas with the empty left side: $\rightarrow B$ that we sometimes will simplify to $B$ by dropping the arrow at the beginning of a formula. We shall not allow formulas with the empty right side. Inference rules are the following:

$$\frac{A \rightarrow B \quad B \cup C \rightarrow D}{A \cup C \rightarrow B \cup D} \qquad (1)$$

$$\frac{A \rightarrow B \cup C}{A \rightarrow B} \qquad (2)$$

The first rule allows us to build a new object $A \cup C \rightarrow B \cup D$ which is called the *conclusion* of the rule, from any two given objects $A \rightarrow B$ and $B \cup C \rightarrow D$, called *premises* of the rule. The second rule has only one premise, and it allows one to "simplify" objects by dropping some of the symbols from the set on the right side of the rule.

This language of objects and inference rules determine a free deductive system. Providing it with an interpretation gives us a knowledge system. Let us assume that meaning of each symbol of the alphabet is some value. In this sense we can consider these symbols as variables. We assume that meaning of an object $A \rightarrow B$ is a function that computes a meaning of $B$ from a meaning of $A$, i.e. the values of all elements of $B$ from the values of elements of $A$.

Let us consider a deductive system of this class with some initial objects, let us say $P \rightarrow Q$, ..., $R \rightarrow S$. Meaning of each initial object is assumed to be given. The meanings of derived objects are constructed from the meanings of given objects in the following way. If the first derivation rule is applied, then the new meaning will be $f;g$, that is a sequential application of functions $f$ and $g$ that are the meanings of the first and the second premise of the rule. If the second rule is applied, then the new meaning is a projection of the meaning of $A \rightarrow B \cup C$ onto $A \rightarrow B$, i.e. it computes only the meaning of $B$.

Let us fill now this knowledge system with the knowledge about rectangular triangles. First, we shall make precise the alphabet. It will contain notations for sides: *a, b, c* and for angles: *alpha* and *beta*. Knowledge will be represented by the following initial objects:

$\{a,b\}\rightarrow\{c\}$, $\{a,c\}\rightarrow\{b\}$, $\{b,c\}\rightarrow\{a\}$, $\{alpha\}\rightarrow\{beta\}$, $\{beta\}\rightarrow\{alpha\}$,
$\{a,c\}\rightarrow\{alpha\}$, $\{a,alpha\}\rightarrow\{c\}$, $\{c,alpha\}\rightarrow\{a\}$.

The interpretation is hidden in the equations:

$a^2+b^2=c^2$
$alpha+beta=\pi/2$
$sin(alpha)=a/c.$

In order to implement the interpretation, we must have some mechanism of equation solving, or otherwise, we can solve the equations for each computable side or angle, i.e. write a program for each initial object for computing the required value, e.g. *c=sqrt(a*a+b*b)* will be the interpretation of $\{a,b\}\rightarrow\{c\}$ e.t.c. Now we can test the knowledge system on an example "compute *a* from given *beta* and *c*". It is possible to do by computing *alpha* from *beta*, and thereafter computing *a* from *c* and *alpha*. Indeed, applying the inference rules, we get the following derivation:

$$\dfrac{\dfrac{\{beta\}\rightarrow\{alpha\} \quad \{c,alpha\}\rightarrow\{a\}}{\{c,beta\}\}\rightarrow\{a,alpha\}}}{\{c,beta\}\}\rightarrow\{a\}.}$$

We see from this example that, in order to specify a knowledge system, one has to do the following:

- specify a language
- give inference rules
- give rules for constructing interpretations of derivable objects.

We expect that interpretations of deductive systems, i.e. the meaning of knowledge, can be of the extreme variety, because this reflects the diversity of the real world. Deductive systems that have in some sense minimal meaning are investigated in logics and there are quite general interesting results, some of which will be briefly reviewed in the following sections.

**Post's systems**. Let us have an alphabet *A*, a set of variables *P*, a set **A** of initial objects that are words in the alphabet *A*. These initial objects are also called axioms**.** Inference rules are given in the following form:

$$\frac{S_1, \ldots, S_m}{S_0},$$

where $S_0, S_1, \ldots, S_m$ are words in the alphabet $A \cup P$, and $S_0$ doesn't contain variables which do not occur in $S_1, \ldots, S_m$. A word $W_0$ is derivable from the words $W_1, \ldots, W_m$ by an inference rule in the calculus if and only if there exists an assignment of words in *A* to variables occurring in the rule, such that it transforms the words $S_0, S_1, \ldots, S_m$ into $W_0, W_1, \ldots, W_m$. This kind of a calculus is called a *Post's system*. These calculi were introduced and

investigated by E. Post already before computers became known as a way to formalize reasoning and computing. They have considerable generality and some very fine properties showing the power of syntactically simple calculi.

If $B$ is included into $A$, and $A$ is the alphabet of a Post's system, then we say that the Post's system is given over $B$, and we can consider the words in the alphabet $B$ generated by the system. The following are some important definitions.

Two Post's systems over $B$ are *equivalent Post´s systems* if and only if they generate the same set of words in $B$.

In the following definitions, $p$, $p_1$, $p_2$ are variables. We have a Post's system *in normal form* if and only if there is only one axiom in it and all its inference rules are in the following form, where $S_1$ and $S_0$ do not contain variables:

$$\frac{S_1 p}{p S_0}.$$

We have a *local Post´s system*, if and only if there is only one axiom in it and its rules are in the form

$$\frac{p_1 S_1 p_2}{p_1 S_0 p_2}$$

A Post's system is a *decidable Post´s sytem*, if and only if there exists a regular way (an algorithm) to decide for any word whether the word is generated by it or not.

A set of words is called *recursively enumerable set* if and only if there exists a Post's system that generates it.

*Example.* Let us define a Post's system that generates arithmetic expressions including identifiers $x$, $y$, $z$ and plus and minus signs. The alphabets are $B=\{ x, y, z , +, -\}$ *and* $A=B\cup\{v\}$. The axioms are $x$, $y$, $z$, and the rules are

$$\frac{pv}{px} \qquad \frac{pv}{py} \qquad \frac{pv}{pz} \qquad \frac{p}{p+v} \qquad \frac{p}{p-v} \ ,$$

where $p$ is a variable.

There are some interesting facts known about Post's systems:

1. It is possible to build an equivalent normal form Post's system for any Post's system.

2. It is possible to build an equivalent local Post's system for any Post's system.

3. There exist undecidable Post's systems (and as a consequence, exist undecidable recursively enumerable sets).

4. There exists a *universal Post's system* that generates exactly the set of all words *NW* such that *N* is a coding of a Post's system and *W* is a word generated by the Post´s system.

## 1.2. Brute force deduction and value propagation

As the first algorithm of knowledge handling, we consider a *brute force search* that generates any given element of a recursively enumerable set *S* represented by a Post's system, where each rule has one premise, e.g. a Post's system is in normal form. This algorithm must generate words in some regular way, so that we can be sure that each word will be generated, if the algorithm works for sufficiently long time. Let us divide the set *S* into subsets $S_1$, $S_2$,... of words that can be generated in one step, two steps etc. These are finite sets, because there is only one axiom to start with and a finite number rules which may be applicable at each step. We shall build our algorithm so that it generates sequentially the sets $S_1$, $S_2$,..., starting from a given set $S_0$ that contains only the axioms of the calculus. This algorithm uses the following entities:

*axioms* - set of axioms of the calculus;
*wanted* - word to be generated;
*active* - set of words usable to produce the new set $S_i$;
*new* - currently built set $S_i$;
*app(r,w)* - function that for a given rule *r* and word *w* produces a word by applying the rule *r* to *w;*
*applicable(r,w)* - *true* iff rule *r* is applicable to word *w.*

The algorithm is as follows:

**A.1.1:**

```
active = axioms;
new = axioms;
        while wanted ∉ active & not empty(new) do
                new={};
                for w ∈ active do
                        for r ∈ rules do
                                if applicable(r,w) then
                                    new=new ∪{app(r,w)}
                                fi
                        od
                od;
                active= new ∪active;
        od;
        success()
```

This algorithm is slow and also uneconomical from the memory requirements point of view. In order to improve the performance, one has to take into account additional knowledge that can be supplied only if the problem is more specific.

Now we continue an example from the section 1.1.2, and build a simple inference algorithm for the calculus of computability. This is a *value propagation algorithm*, and it is a concretization of the brute-force algorithm A.1.1. It is intended for deriving objects of

the form *in* $\rightarrow$ *out* that express the goal: "find a function for computing elements of *out* from elements of *in.*" The algorithm repeatedly searches through all axioms and applies the axioms that extend the set of computable elements.

We will use the following notations in this algorithm:

*known* - set of elements already known to be computable;
*out* – set of elements to be computed;
*found* - boolean variable showing whether something was found during the last search through all axioms;
*axioms* - set of axioms;
*arg(r)* - set of given elements in axiom *r,* i.e. the elements on the left side of the arrow;
*res(r)* - set of computable elements in axiom *r*;
*plan* - sequence of applied axioms which is a plan for computations.

**A.1.2:**

```
known=in;
plan:=();
while not out ⊆ known do
        found=false;
        for r ∈ axioms do
                if arg(r) ⊆ known & res(r) ⊄ known then
                        found=true;
                        known=known ∪ res(r);
                        append(plan,r)
                fi;
        od;
        if not found then failure() fi
od;
success(plan)
```

It is easy to see that the algorithm A.1.2 requires no more than $n^2$ search steps for $n$ axioms. Indeed, the number of steps for finding an applicable axiom is at most $n$, and there are at most $n$ applicable axioms in the composed plan. We say that the time complexity of this algorithm with respect to the number of axioms $n$ is $O(n^2)$. This algorithm constructs a plan that may contain excessive steps. Better algorithms, which work in linear time and produce minimal plans, will be described in the section for constraint propagation.

## 1.3. Clausal calculi and resolution

### 1.3.1. Language

Post's calculi gave us some interesting general results about the derivability in general. We believe that, in principle, they can be used for representing any kind of knowledge. But they are very impractical for representing any particular kind of knowledge, since they are too universal. Another rather general knowledge representation and handling formalism is logic. One of the most popular logical languages for knowledge representation is the language of clauses that we describe here.

We use *expressions* in this language that represent computations and denote objects, they are called *terms*. We use *atomic formulas* for representing relations between objects and, finally, we can collect atomic formulas into *clauses* that represent logical knowledge - selections of alternatives. Any meaningful piece of knowledge will be presented as a set of clauses in this language.

*Terms* are built from variables and function symbols. We can use symbols of functions with zero arguments as constants.

Examples: *sin(alpha),*
*x+2*y,*
*length_of(a).*

*Atomic formula* is of the form $P(e_1,...,e_n)$ where $P$ is a predicate symbol, denoting some $n$-ary relation (i.e. a relation which binds exactly $n$ objects) and $e_1,...,e_n$ are terms.

Examples: *father(Tom,John),*
*employee(x),*
*less_than(a,b+a).*

Atomic formula is negated (occurs negatively), if it has a minus sign, for instance *-father(Tom,John).* It's meaning is the opposite to the meaning of the formula without minus.

*Clause* is a collection of atomic formulas and their negations. It is convenient to speak about a clause as a set of *literals*, which are nonnegated atomic formulas called *positive literals*, or negated atomic formulas called *negative literals*. Meaning of a clause consisting of literals $L_1,...,L_n$ is: "There are only the options $L_1,...,L_n$."

Examples: *{A,-B},*
*{-father(x,y),parent(x,y)},*
*{-parent(x,y), father(x,y),*
*mother(x,y)},*
*{}.*

The latter is an empty clause, which means absence of options, i.e. the impossibility or falsity.

If literals contain no variables, they are called *ground literals*; if a clause contains only ground literals, it is called a *ground clause*. A special case of clauses are *Horn clauses*, which contain at most one positive literal each. Horn clauses are sometimes represented in another form that shows the positive literal positionally at the point of the arrow. For example

$A, ..., B \rightarrow C$ or $C \leftarrow A, ..., B,$

where $C$ is the positive literal, are two different representations of one and the same Horn clause

*{-A,…,-B,C}.*

## 1.3.2. Inference rule – resolution

Let us consider the following two clauses *{A}* and *{-A}*. If these two clauses appear together, then we can conclude that there are no options at all, i.e. there is a contradiction, because the first tells us "There exists only the option *A*," but the second tells us: "There exists only the option *not A*." We can generalize this reasoning in two ways. Firstly, if the two clauses contain other literals besides *A* and *-A*, we can conclude that *A* and *-A* can be dropped. We say that the literals *A* and *-A* form a *contradictory pair*. This gives us the following *simple resolution rule*:

$$\frac{\{A,B,...,C\} \qquad\qquad \{-A,D,...,E\}}{\{B,....,E\}}$$

where *A* is an atomic formula, *B,...,C,D,...,E* are literals. The clause *{B,...,E}* contains all elements from *B,...,C* and *D,...,E*, but doesn't contain multiple elements or elements of the contradictory pair. This clause is called the *resolvent* of premises of the rule.

The other generalization is more complicated. It concerns unification that will be considered in details in the following chapter. Let us consider an example first. Let us have two slightly different literals with the same predicate name, one negative and the other positive: *-cat(Tom)* and *cat(x)*. We still can apply the reasoning we did for constructing the resolution rule, because these literals, although different, can be transformed into a contradictory pair. This can be shown by saying that the variable *x* can be equal to *Tom,* and this gives a contradiction. Let us consider once again what we have done here. We have found a substitution *x/Tom* which gave us the literals *cat(Tom)* and *-cat(Tom)*, when applied to literals c*at(x)* and *-cat(Tom)*. This procedure is called *unification*. In general, a *substitution* is a tuple of pairs *(variable, term)*. Application of a substitution *s* to a term or a formula *F* is simultaneous changing of the variables in *F* to their corresponding terms in the substitution *s*. We denote the result of this application by $F^{\circ}s$. A substitution *s* that gives $F1^{\circ}s=F2^{\circ}s$ is called a *unifier* for *F1* and *F2*, and its application to *F1* and *F2* is called *unification*.

*General resolution rule* is the following:

$$\frac{\{A,B,...C\} \qquad\qquad \{-A',D,...,E\}}{\{B^{\circ}s,...,E^{\circ}s\}} \quad A^{\circ}s=A'^{\circ}s$$

where *s* is a unifier for *A* and *A'* and the resolvent $\{B^{\circ}s,...,E^{\circ}s\}$ contains results of applying the substitution *s* to literals *B,..., C, D,..., E* without repetition of identical literals and without contradictory pairs.

Clausal calculi have become very popular, because they are handy to use and they also have a nice and neat logical explanation. Originally, the resolution method was developed in 1965 by J. A. Robinson just for automation of logical deductions [43]. At the same time and independently, S. Maslov developed an inverse method of deduction in logic that was based on the same principle [30].

Given a finite set of clauses, the resolution method allows us to answer the question, whether this set of clauses is unsatisfiable (contradictory), i.e. whether there exists a derivation of the empty clause from the given set of clauses. Derivation of an empty clause is called *refutation.* This is the most common usage of resolution. If one wishes to prove something presented as a literal, then he/she can refute the negation of this literal. This is a general way of using resolution in theorem-proving.

### 1.3.3. Resolution strategies

There are several useful *resolution strategies* that take into account additional information and speed up the derivation process.

*1. Deletion strategy.* A literal is pure (or passive) in a set of clauses, if there is no literal in the set of clauses that can constitute a contradictory pair with the given literal. A literal is pure, for instance, if there are no other literals with the same predicate symbol, or the literals with the particular predicate symbol are either all positive or all negative in the clauses. Clauses that contain pure literals can be obviously ignored in the derivation of the empty clause.

If a clause contains two literals that differ only by their signs, this clause can be removed from the set of literals without influencing the results of derivation. Such a clause is called a *tautology*.

A clause C1 *subsumes* the clause C2 iff there exists a substitution *s* such that C1°*s* is included into C2. Subsumed clauses can be eliminated, because the set of clauses remaining after the elimination is satisfiable if and only if the original set is satisfiable.

2. *Unit resolution. Unit resolvent* is a resolvent for which at least one of the premises is a *unit clause* - contains one literal. Unit resolution is the process of deduction of unit resolvents. It is complete for Horn clauses, and good algorithms are known for this case.

3. *Input resolution. Input resolvent* is a resolvent for which at least one of the premises is in the initial set of clauses. Input resolution is the process of deduction of input resolvents. It is complete for Horn clauses, and good algorithms are known for this case.

4. *Linear resolution.* A linear resolvent is a resolvent for which at least one of the premises is either in the initial set of clauses or is an ancestor of another premise of this resolution step. Linear resolution is complete for refutation (deriving an empty clause).

5. *Set of support resolution.* A subset *E* of the set of clauses *D* is called a *support set* of *D*, iff *D**E* is satisfiable. A *set of support resolvent* is a resolvent for which at least one of the premises is either in the set of support or is its descendant. Set of support resolution is the process of deduction of set of support resolvents. It preserves completeness, and may be very efficient, if the support set is well constructed. The support set can represent a goal in the deduction process, i.e. it makes the resolution goal-directed.

6. *Ordered resolution*. Literals are ordered in clauses from left to right. The ordering of literals of premises is preserved in resolvents. Only first, i.e. the leftmost literals in clauses can be resolved upon. This is a restrictive strategy, but it is still complete for Horn clauses.

There are more strategies known for improving the resolution process, and even the strategies listed above can be adapted and developed further.

## 1.4. Pure Prolog

*Prolog* is a programming language developed in seventies of the last century. A good textbook for Prolog programming is [6]. We consider here pure Prolog that does not include built-in predicates or any other non-logical extensions. A program in pure Prolog

in essence includes knowledge about the problem that has to be solved. The solution process is very much knowledge processing with the aim to solve the given problem. We present a very simple example of a program in Prolog here. Each line of the program is a Horn clause. The combination of symbols *:-* is used instead of the arrow ←. The goal of computation is written after the symbols *?-*. Identifiers beginning with capital letters are variables, other are constants. A program that is written in pure Prolog is just a set of Horn clauses, for example:

```
parent(john,judy).
parent(judy,bob).
parent(judy,pat).
parent(ann,tom).
parent(tom,bob).
grandparent(X,Z) :- parent(X,Y), parent(Y,Z).
?-grandparent(X,bob).
```

This program includes a description of a family tree (that may contain much more knowledge than it is needed for solving the particular problem) and a clause that defines grandparent. This is sufficient for finding the answer that is

*X = john;*
*X = ann;*

Indeed, we have a logical proof for the first answer in Fig. 1.1 (and a proof for the second answer can be found in a similar way):

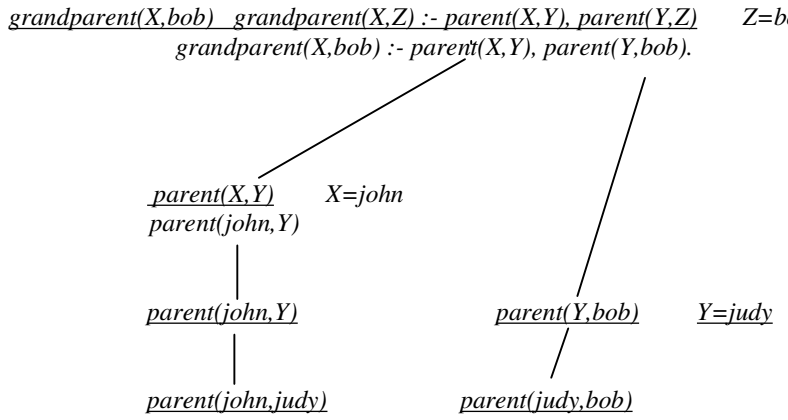



**Figure 1.1.** Derivation in Prolog

Execution of program that is written in pure Prolog is application of the ordered linear unit resolution. Let us assume that the positive literal in a clause of a Prolog program is always the leftmost literal in the clause. A program is executed with a goal which is a unit clause containing only a negative literal, but in the process of solving the problem new subgoals may appear.

Let us have the following notations:

*prog* - program to be executed;
*goals* - list of goals, which initially contains only the *goal* given by a user;
*successful* - a variable that signals whether *success* or *failure* occurred;
*unifier(x,y)* - produces the most general unifier of *x* and *y*, if such exists, and *nil* otherwise;
*apply(x,L)* - applies the unifier *x* to the list *L*.

A *Prolog interpreter* is represented by the following recursively defined function *exec(prog,goals,successful)*. It takes a program *prog*, and set of goals *goals* and assigns a value *true* or *false* to the parameter *successful.* This function prints out the atoms that unify with the initial goal. Printing is not explicitly shown in the algorithm – printing is hidden in the function *success().*

**A.1.3:**

```
exec(prog,goals,successful)=
if empty(goals) then success( )
 else goal= head(goals);
        goals= tail(goals);
        L: {rest= prog;
            while not empty(rest) do
                U=unifier(goal,head(head(rest));
                if U ≠ nil then
                        goals1= (apply(U,tail(head(rest)));
                        exec(prog,goals1,successful);
                        if successful then exit L fi
                fi;
                rest=tail(rest);
            od;
            failure( )
            };
        exec(prog,goals, succes)
fi
```

We give some hints to understand this program. The main loop is in a block with the label *L*. It tries to solve a leftmost goal by trying to unify it with the head (with the leftmost literal) of a first clause in the part of the program that is not yet used for this goal. This is represented by the expression *unifier(goal,head(head(rest)).* If the unifier exists, then it is applied to all negative literals *tail(head(rest)* of the clause *head(rest)* and the resulting literals are introduced as new goals *goals1*. Then the process continues by recursively performing *exec(prog,goals1,successful)*. If the unification is unsuccessful, then the clause is dropped and the process continues with the remaining clauses from the remaining part of the program that is *rest=tail(rest).*

## 1.5. Nonmonotonic theories

Knowledge may be nonmonotonic. i.e. some existing facts may become invalid when the amount of knowledge used in derivations is increased - this can be seen from the following simple example. The clauses

*bird(x)* $\rightarrow$ *flies (x)*
*bird(Tweety)*

give us

*flies(Tweety).*

However, if more facts are obtained, for instance, if it becomes known that

*penguin(Tweety)*

we must discard the derived fact

*flies(Tweety).*

There are various ways of handling *nonmonotonic knowledge*, see Genesereth and Nilsson [11]. We shall describe here calculi that have besides conventional inference rules also *defaults* that are applicable under certain conditions called *justifications*. These calculi are called *default theories*.

A default has the following form

$$\frac{A:B1, \ldots, Bk}{C},$$

where the formula *A* is a premise, the formula *C* is a conclusion and the formulas *B1, ..., Bk* are justifications. Conclusion of the default can be derived from its premise, if there is no negation of any justification derived. We can say that, in this case, we can believe that the justifications hold. As soon as it becomes clear that a negation of some justification has been derived, then the derivation of the conclusion of the default becomes invalid. This gives us the nonmonotonicity. The knowledge about birds in the example from the beginning of the present section can be represented by the following default:

$$\frac{bird(x): flies(x)}{flies(x)}$$

The following rather simple schema of defaults

$$\frac{:not\ F}{not\ F}$$

where *F* is any ground formula of a theory, represents *closed world assumption* (CWA). This assumption means that any ground formula which is not derivable in the theory is considered to be false. One must use this assumption cautiously. For example, having the clause *{P(x),Q(x)},* we still must assume under CWA *not P(A)* and *not Q(A)* for any constant *A*, if neither of the formulas *P(A)* and *Q(A)* is derivable.

Application of defaults requires in general more efforts than application of an inference rule, because one must check that no justification is being disproved. The function *defaultDerivation(A,C,r)* is applicable for a derivation step with a default.

*A* - premise of a default;
*C* - conclusion of a default;
*r* – default;
*justify(A,C,r)* – a function that produces the set of justifications needed for deriving *C* from *A* from the default *r*.

**A1.4:**

```
defaultDerivation(A,C,r)=
        J=justify(A,C,r);
        for B ∈ J do
                if not B then failure() fi
        od;
        success()
```

Deciding about ***not B*** may need much computations inside the presented loop, and this increases the complexity of the derivation with defaults.

## 1.6. Production rules

*Production rules* are a well-known form of knowledge that is easy to use. A production rule is a condition-action pair of the form

*(condition, action)*

with the meaning: "If the *condition* is satisfied, then the *action* can be taken." Also other modalities for performing the action are possible - "must be taken", for instance. Both the condition and the action can be presented in some programming language, this makes it easy to implement a knowledge system with productions rules. Production rules are a general and flexible form of representing procedural knowledge. It is easy, for instance, to give an algorithm that transforms any sequential program into the form of production rules. In order to do this, one has to assign a logical variable to each program point where some action has to be taken. Values of these variables must be computed in such a way that only the variable of a program point, where the next action has to be taken, has the value *true*. Values of these variables must be recomputed after each action, more precisely, each action of the program must be extended with the evaluation of its logical variable (that becomes *false*) and of logical variables of the actions that immediately follow the given action.

Let us have a set of production rules called *Productions* and functions *cond(p)* and *act(p)* which select the condition part and action part of a given production rule *p* and present them in the executable form. The following is a simple algorithm for problem solving with production rules:

**A.1.5:**

```
        while not good() do
                found = false;
                for p ∈ Productions do
                        if cond(p) then
```

```
                        act(p);
                        found=true
                fi
        od;
if not found then failure() fi
od
```

The results of applying this algorithm may depend on the ordering of rules. This enables a user to control the process of problem solving, but decreases the generality of the knowledge. In particular, it complicates the usage of rules as an internal language hidden from the user.

One can extend production rules by adding *plausibility* values to them. Let us associate with each production rule $p$ a plausibility $c(p)$ of application of the production rule. These values can be in the range from 0 to 1. We shall consider as satisfactory only the results of application of a sequence of rules $p, ..., q$ for which the plausibilities $c(p) ,..., c(q)$ satisfy the condition $c(p) * ... * c(q) \geq cm$, where $cm$ is the minimal satisfactory plausibility of the result. When selecting a new applicable production rule, it is reasonable to select a production rule with the highest value of plausibility. This is done by the for-loop before deciding about applicability of the selected rule. The selected rule is always the best rule for application, because it has the highest plausibility. This gives us the following algorithm for solving problems with a production system with plausibilities:

**A.1.6:**

```
c=1;
while not good() do
                x=0;
                for p ∈ Productions do
                        if cond(p) and c(p) > x then
                                p1=p;
                                x=c(p)
                        fi
                od;
                c=c*x;
                if c ≥ cm then act(p1) else failure() fi
od;
success()
```

This algorithm has also the advantage that the result of its application doesn't depend on the order of production rules in the set *Productions*.

Plausibility of a situation arising after applying a production rule can be computed in a variety of ways, not only by multiplication as in the algorithm presented above. We get a general algorithm of application of production rules, if we introduce a new function *plausibility(x,y)* that computes a plausibility of a new situation arising after application of a rule $x$ in a situation with plausibility $y$. We assume in the algorithm below that there are always rules that have plausibility higher than 0, hence the for-loop in the algorithm will always select some rule *p1*.

**A.1.7:**

```
c=1;
while not good() do
```

```
                x=0;
                for p ∈ rules do
                    if cond(p) and plausibility(p,c) > x then
                        p1=p;
                        x= plausibility(p,c)
                    fi
                od;
                if x ≥ cm then
                    act(p1);
                    c=x
                else failure()
                fi
od;
success()
```

## 1.7. Decision tables

A compact form of representation of a small number of production rules are *decision tables*. A decision table consists of three kinds of subtables: table of conditions, selection matrix and table of values. Table of conditions includes atomic conditions (predicates) for selecting production rules. Selection matrix combines them into complete conditions for selecting a rule by showing whether an atomic condition or its negation is required. Table of values contains results of a selection that can be either values or actions. Fig. 1.2 shows the structure of a 2-dimensional decision table that contains two tables of conditions and two selection matrices. Only selection of one decision is shown in the selection matrices in Fig.1.2,where *y* and *n* denote that a condition (respectively its negation) must be satisfied.

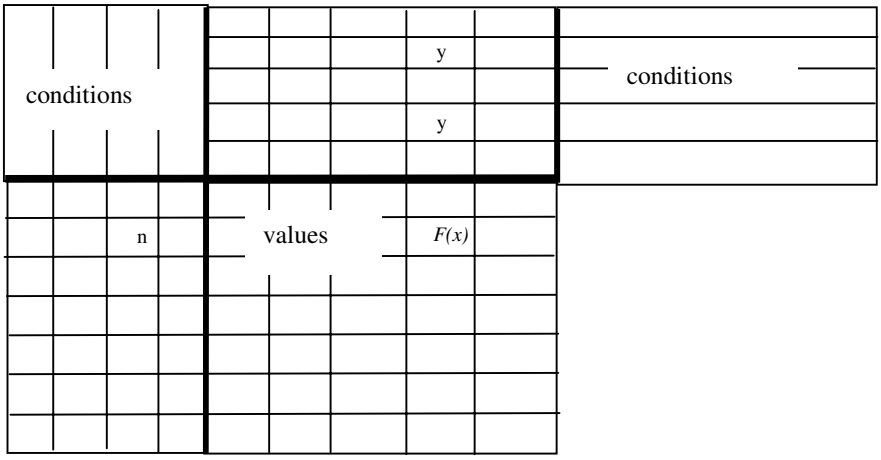



**Figure 1.2.** Decision table

## 1.8. Rete algorithm

*Rete algorithm* has been developed for fast search of applicable rules in a large rule base. It uses a specific data structure, let us call it *Rete graph* for improving the accessibility of rules. It is used in JESS (Java Expert System Shell) and in its predecessor – CLIPS. We shall consider this algorithm in three parts:

- knowledge representation,
- knowledge management (i.e. introduction of changes into the knowledge base)
- knowledge usage.

For the Rete algorithm, knowledge is represented in knowledge triplets that include three pieces: *(predicate, arg, arg)*. A *knowledge triplet* can represent three kinds of knowledge:

1.facts, e.g.
*(goal e1 simplify), (goal e2 simplify), (goal e3 simplify),*
*(expr e1 0+3), (expr e2 0+5), (expr e3 0*2)*

The meaning of a fact may vary, as we see from the example. The first fact above represents a goal that means "simplify the expression *e1*". The fourth fact above says that there is an expression *e1* and it is *0+3 etc.* Facts are ground formulas.

2. patterns, e.g.
*(goal ?x simplify)*
*(expr ?y 0?op ?a2)*
*(parent ?x ?y)*

A pattern includes variables – identifiers beginning with question mark. A pattern is in essence an atomic formula with variables.

3. rules, e.g.
*(R1 (goal ?x simplify) (expr ?x 0+?y) => (expr ?x ?y))*
*(R2 (goal ?x simplify) (expr ?x 0*?y) => (expr ?x 0))*

Rules, as usual, include a condition part and an action part. The condition part of a rule is conjunction of atomic formulas presented as patterns. Action part of a rule can be a description of a transformation.

Patterns, rules and facts are composed in an acyclic graph with separate layers for predicate names, patterns, relations and rules. Relations show possible evaluations of variables. Fig. 1.3 shows a general structure of a Rete graph with layers for predicate names, patterns, relations and rules. The nodes of patterns are called *alpha-nodes*, and the nodes of binary relations are called *beta-nodes*. Fig. 1.4 shows a Rete graph for the facts, patterns and rules given above.

It is easy to see that search of a rule applicable to a fact, or of a proper place for a new fact will proceed down from the root, first by selecting the right predicate name, then by selecting a suitable pattern and thereafter by finding nodes where a relation matches the pattern and the fact.

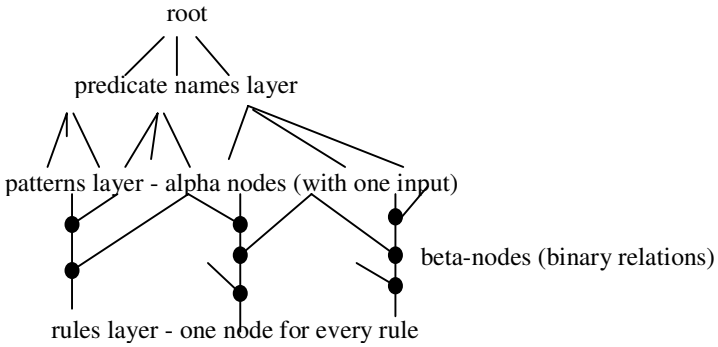



**Figure 1.3.** Structure of a Rete graph

Let us consider now knowledge handling in a Rete graph. It may be extending the graph or using it for selection of applicable rules. When a new predicate occurs, it is added to the predicate layer. When a new pattern occurs, it is added to the pattern layer. When a fact arrives, then the following actions will be taken:

- select a predicate
- select a pattern and take it as a current node
- find a relation that matches the pattern among the successors of the current node, update its relation by adding a new line, take it as the new current node, and repeat this until all parts of the fact are included in the relations.

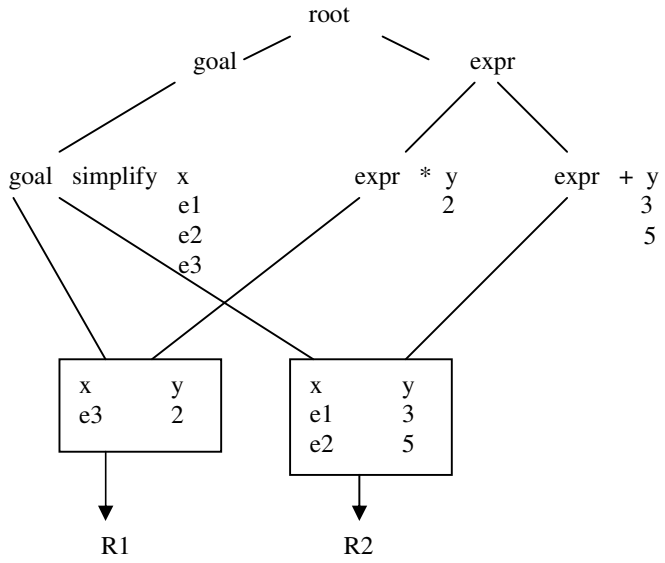



**Figure 1.4.** Example of Rete graph

When a goal arrives, although it is also in the form of a fact, the following actions will be taken:

- select a predicate
- select a pattern and take it as a current node
- find a relation that matches the pattern among the successors of the current node, take it as the new current node and repeat this until a rule is reached
- apply the rule.

One can see that search in a Rete graph for adding a fact and applying a rule is almost the same. Let us consider here an algorithm of the search of an applicable rule. This algorithm can be presented using the following notations:

*root* – root of a given Rete graph;
*predicates* – set of predicates in the predicate layer of the given Rete graph
*patterns* – set of patterns in the pattern layer of the given Rete graph;
*successors(x)* – successors of the given node *x;*
*matches(beta,fact)* – true, if the node *beta* matches the *fact;*
*isRule(currentNode)* – true, if *currentNode* is a rule;
*pred(fact)* – returns predicate of *fact;*
*apply(rule)* – applies the rule *rule.*

The algorithm A.1.8 below is a *Rete search algorithm* that finds the right rule for the given *goal,* and applies the rule. If a predicate or a pattern can not be found, then it signals by calling *failure().* This algorithm uses a function *findRule(currentNode,fact)* that searches for the applicable rule, starting from the *currentNode* that in the beginning of search is a pattern node, but later is a beta node on the path to the rule. The first loop (*L1*) is for finding a predicate. The second loop (*L2*) finds a pattern *pat* and then applies the function *findRule(pat,fact)* for finding the right rule and applies the rule.

**A.1.8:**

```
rete(goal)=
    x=root;
    found=false;
    L1: for pred∈predicates do
            if predicate==pred(goal) then
               found=true;
               x=predicate;
               break L1
            fi
    od;
    if not found then failure() fi;
    L2:for pat∈patterns do
            if applicable(pat,goal) then
                    rule=findRule(pat,goal);
                    if rule==nil then
                            failure()
                    else
```

```
                    apply(rule);
                    success()
                fi
            fi
        od;


findRule(currentNode,fact)=
            for beta∈ successors(currentNode) do
                if matches(beta,fact) then
                    currentNode=beta;
                    findRule(currentNode,fact)
                fi
            od;
            if isRule(currentNode) then
                return currentNode
            else
                return nil
            fi
```

It is possible to see that the function *findRule(currentNode,fact)* performs a depth-first search on beta nodes of a Rete graph – it goes recursively down on the Rete graph.

## 1.9. Semantic networks

Linguists noticed long ago that structure of a sentence can be represented as a *semantic network*. Words of the sentence are nodes, and they are bound by arcs expressing relations between the words. The network as a whole expresses a meaning of the sentence in terms of meanings of words and relations between the words. This meaning is an approximation of the meaning that people can assign to the sentence. Let us take the following sentence as an example:

*John picks up his report in the morning and has a meeting after lunch. After the meeting he will give the report to me.*

A semantic network representing the meaning of this sentence is shown in Fig. 1.5.

Inferences can be made, depending on the properties of the relations of a semantic network. Let us consider only time relations of the network in our example, and encode the time relations by atomic formulas as follows:

*before(lunch,morning)*
*after(morning,lunch)*
*after(lunch,have a meeting)*
*after(have a meeting,give)*
*at-the-time(morning,pick up)*

We could continue encoding of the semantic network in the same way, representing each arc by an atomic formula.

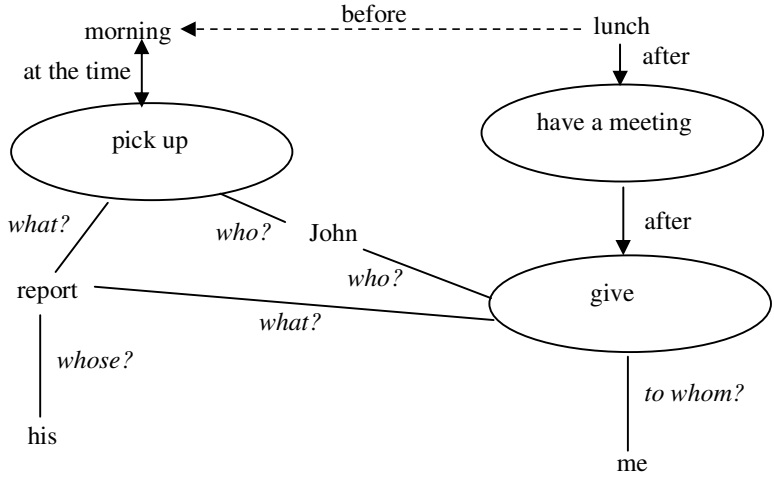



**Figure 1.5.** Semantic network

Inferences on semantic networks are done by propagating facts (more generally - "pieces of knowledge") along the arcs of he network. As a result, the network is changed. Inference rules, as usual, are schemes of the following form:

$$\frac{S1, \ldots, Sm}{S0}$$

describing derivation of a conclusion *S0* from premises *S1,...,Sm*. In our example, rules for using time-dependencies are the following:

$$\frac{before(x,y) \quad before(y,z)}{before(x,z)}$$

$$\frac{after(x,y)}{before(y,x)}$$

$$\frac{at\text{-}the\text{-}time(x,z) \quad before(y,z)}{before(y,x)}$$

Applying these rules, we can derive

$$\frac{after(lunch,have\ a\ meeting)}{before(have\ a\ meeting,lunch)}$$

and

$$\frac{at\text{-}the\text{-}time(pick\ up,morning) \quad before(lunch,morning)}{before(lunch,pick\ up)}$$

etc.

## 1.10. Frames

Marvin Minsky postulated in 1974 a hypothesis that knowledge can be represented in bundles which he called *frames* [35] and he described informally essential properties of frames:

- The essence of a frame is that it is *a module of knowledge* about something that we can call a concept. This can be a situation, an object, a phenomenon, a relation.
- Frames contain smaller pieces of knowledge: *components, attributes, actions* that can be (or must be) taken when conditions for taking an action occur.
- Frames contain *slots* that are places to put pieces of knowledge in. These pieces may be just concrete values of attributes, more complicated objects, or even other frames. A slot is filled when a frame is applied to represent a particular situation, object or phenomenon.

Knowledge representation languages, e.g. KRL and FRL, were developed for representing knowledge in the form of frames. An essential idea developed in connection with frames was *inheritance*. Inheritance is a convenient way of reusing existing knowledge in describing new knowledge. Knowing a frame *f*, one can describe a new frame as a kind of *f*, meaning that the new frame inherits the properties of *f*, i.e. it will have these properties in addition to newly described properties described. Inheritance relation expresses very precisely the relation between super- and subconcepts. Fig.1.6 contains a small inheritance hierarchy as an example.

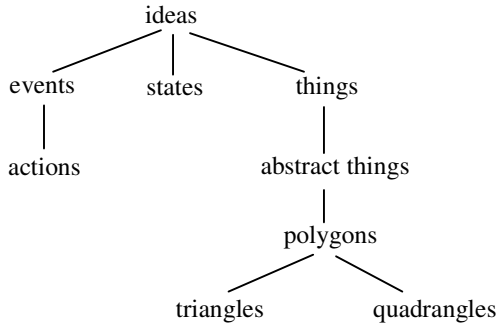



**Figure 1. 6**. Example of inheritance hierarchy

There are frame languages for describing a special kind of frames – *computational frames* [48]. These frames bear information about computability of objects. This knowledge can be used in construction of programs, for example. Briefly, a language of computational frames includes the following kinds of statements:

1) Declaration of variables

*type id[,id,...];*

where the *type* can be either a primitive type or a frame.

2) Binding

*a = b ;*

is an equality, where *a, b* are either variables or components of some structured variables. (A component *x* of a variable *y* is denoted by *y.x*.)

3) Axiom

*precondition* –> *postcondition*{*implementation*}**;**

Axioms are written in a logical language. One can work with different logical languages. The choice depends on the way of usage of frames, in particular, on the availability of a prover to be used in program synthesis. It is easy to handle axioms with preconditions that are conjunctions of the following: propositional variables and implications of conjunctions of propositional variables. The postconditions are disjunctions of conjunctions of propositional variables. Name of any variable can be used as a propositional variable. In such a case, a name denotes that value of the variable can be computed. An implication in a precondition denotes a subgoal – a computational problem whose algorithm has to be synthesized before the method with the precondition can be applied. This logic has been tested in several practically applied synthesizers, in particular, in the CoCoViLa system [16].

Implementation shown at the end of an axiom is a name of a method of the class that implements the computational part of the frame. If the implementation is missing, then the formula states a new subgoal, i.e. a computational problem. In the latter case the formula can have only conjunctions of propositional variables as pre- and postconditions.

4) Equation

*exp1 = exp2 ;*

Equation is given as an equality of two arithmetic expressions *exp1* and *exp2*. Equations are especially useful for gluing components together and adjusting their data representation (units etc.). Which expressions are acceptable in equations depends on the equation solver that has to be a part of the supporting software.

The following are examples of computational frames for square and circle:

```
Square:
    int a,S, d, p;
    S = a*a;
    d*d = 2*a*a;
    p= 4*a;

Circle:
    int S,d,r,p;
    S=3.14*r*r;
    d=2*r;
    p=3.14*d;
```

Having these frames, it is easy to specify computational problems about circles and squares, for example, finding a difference of areas of circle and square shown in Fig. 1.7. A computational frame that specifies this problem is as follows:

*Problem1:*
*Circle c;*
*c. d=1.5;*
*Square s;*
*s. a=C.d;*
*int x;*
*x=s.S –c.S;*
*-> x{};*

Solving the goal ->*x* automatically gives us the desired result.

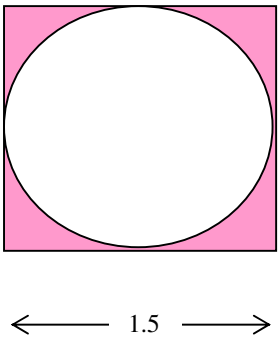



**Figure 1.7.** A computational problem.

## 1.11. Knowledge architecture

In this section we consider knowledge systems introduced in Section 1.1 as modules of larger knowledge-based tools, discuss how knowledge systems interact with one another, and how *knowledge architecture* can be represented by knowledge systems connected in various ways. More about this topic is written in [48].

### 1.11.1. Hierarchical connection

Let us have two knowledge systems $K_1$, $K_2$ with sets of notations (sets of knowledge objects) $S_1$, $S_2$, sets of meanings $M_1$, $M_2$ and notation-denotation relations $R_1$, $R_2$ respectively.

**Definition.** We say that knowledge systems $K_1$ and $K_2$ are *hierarchically connected knowledge systems,* iff there is a relation $R$ between the set of meanings $M_1$ and the set of notations $S_2$, and *strongly hierarchically connected knowledge systems*, iff there is a one-to-one mapping C between the elements of a subset of $M_1$ and a subset of $S_2$.

A hierarchical connection of knowledge systems can be observed quite often. This situation arises, for instance, when we have a knowledge system $K_1$ that is a metasystem of another knowledge system $K_2$, and handles knowledge about the knowledge objects of $K_2$. Its meanings are the knowledge objects of $K_2$.

Having a sequence of knowledge systems $K_1, K_2,..., K_n$, one can build a hierarchy (a tower) of knowledge systems – a *knowledge tower* where each pair $K_i, K_{i+1}$ are hierarchically connected. Towers of closed knowledge systems have been investigated by Lorents in [25]. Practical examples of knowledge towers can be found in networking – stacks of network protocols are examples of towers of knowledge systems.

Let us introduce graphical notations for knowledge systems. Fig. 1.8(a) shows a graphical view of a knowledge system. We have decided to denote explicitly the notations and the denotations (meanings) of a knowledge system. This allows us to show different ways of connecting knowledge systems. Fig. 1.8(b) shows us hierarchical connection of knowledge systems as defined above – a double line connects meanings of $K_1$ with notations of $K_2$. Strong hierarchical connection is shown in Fig. 1.8(c).

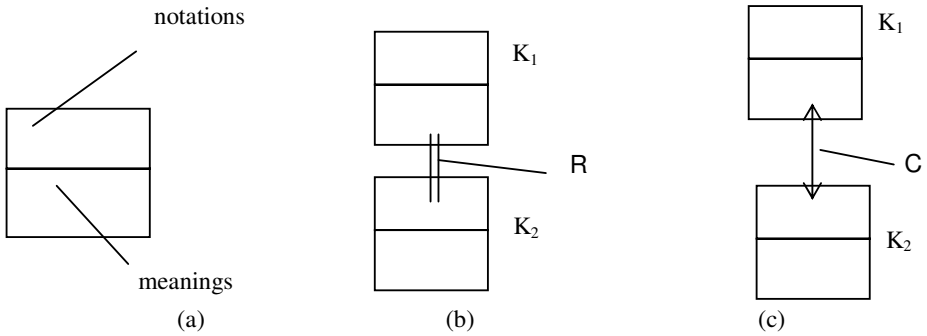



**Figure 1.8.** Knowledge system (a), hierarchical connection (b) and strong hierarchical connection (c) of knowledge systems.

## 1.11.2. Semantic connection

Several knowledge systems can have one and the same set of meanings. This is the case, for instance, with systems of classical logic that have different sets of inference rules. Also programming languages for one and the same computer, and even natural languages for identical cultures have the same sets of meanings. (However, strictly speaking, the cultures depend to some extent on the language, hence there are no absolutely identical cultures with different languages.)

**Definition.** We say that knowledge systems are *semantically connected knowledge systems*, if they have one and the same set of meanings, and then we use the graphical notation as in Fig. 1.9(a). If we wish to express the properties of a knowledge system visually more precisely, then we can add graphical notations inside its boxes. For weak and

closed knowledge systems defined in Section 1.1 the notations are shown in Fig. 1.9(b) and Fig. 1.9(c) respectively.

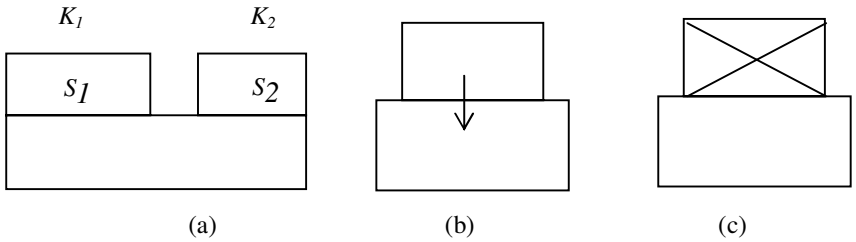



**Figure 1.9.** Semantically connected (a), weak (b) and closed (c) knowledge systems

## 1.11.3. Union

Like for formal languages, one can define a variety of operations (e.g. union and intersection) between semantically connected knowledge systems. For instance, union of semantically connected knowledge systems is defined as follows.

*Union of semantically connected knowledge systems* $K_1$, $K_2$ over a set of meanings $M$ is a knowledge system over the same set of meanings $M$ with the set of object $S_1 \cup S_2$, the notation-denotation relation $R_1 \cup R_2$ and derivation rules of both knowledge systems $K_1$ and $K_2$.

It is desirable to build a union of knowledge systems $K_1$, $K_2$ even when their sets of meanings are different. We can build a new set of meanings $M = M_1 \cup M_2$, and take it as a set of meanings for both knowledge systems. Now we have to add new notations into sets $S_1$ and $S_2$ to denote the added meanings. We have to be careful, so that the derivation rules could not be applied to wrong notations. This is more precisely expressed by the following definitions.

*Conservative extension of knowledge system* $K$ with set of notations $S$ and set of meanings $M$ on a set of meanings $M'$, $M \subset M'$, is a knowledge system with set of meanings M', set of notations $S \cup \{something\}$, where *something* is a new notation that denotes every element of the set $M' \setminus M$, derivation rules of $K$, and notation-denotation relation of $K$ extended by new pairs *(something,x)* for every $x \in M' \setminus M$.

**Theorem.** An object $z$ is derivable from objects $x,...,y$ in a conservative extension $K'$ of a knowledge system $K$ iff it is derivable from the same objects in $K$.

Proof: The derivation rules of $K$ are neither applicable to *something,* nor can they produce *something*, hence the derivability in $K'$ is the same as in $K$.

**Remark.** The notation *something* can denote many things – it introduces an open world. However, one can reason only about the known part of the world, and does not reason about the meanings that are not distinguishable in the knowledge system.

**Definition.** *Union of knowledge systems* $K_1$, $K_2$ is union of semantically connected conservative extensions of $K_1$, $K_2$ on $M_1 \cup M_2$.

It is not our aim here to investigate the properties of knowledge systems built by means of operations like union, intersection etc. that is an interesting area of research in formal languages and systems. We are not going to look deep inside the knowledge systems, but describe the architectures where knowledge systems appear as components, and we investigate only some very general properties of these architectures.

## 1.11.4. Operational connection

Knowledge system $K_1$ is *operationally dependent* on a knowledge system $K_2$, if some of its derivation rules use $K_2$ in deriving a new object, i.e. the result of derivation in $K_1$ depends on knowledge processing in $K_2$.

**Definition.** Knowledge systems $K_1$, $K_2$ are *operationally connected knowledge systems*, if $K_1$ is operationally dependent on $K_2$, and $K_2$ is operationally dependent on $K_1$.

Union of knowledge systems can be sometimes used to specify their operational connection. It can be done, if their sets of meanings have common elements. Fig. 1.10(a) shows that $K_1$ is operationally dependent on $K_2$, and Fig.1.10(b) shows that $K_1$ and $K_2$ are operationally connected knowledge systems.

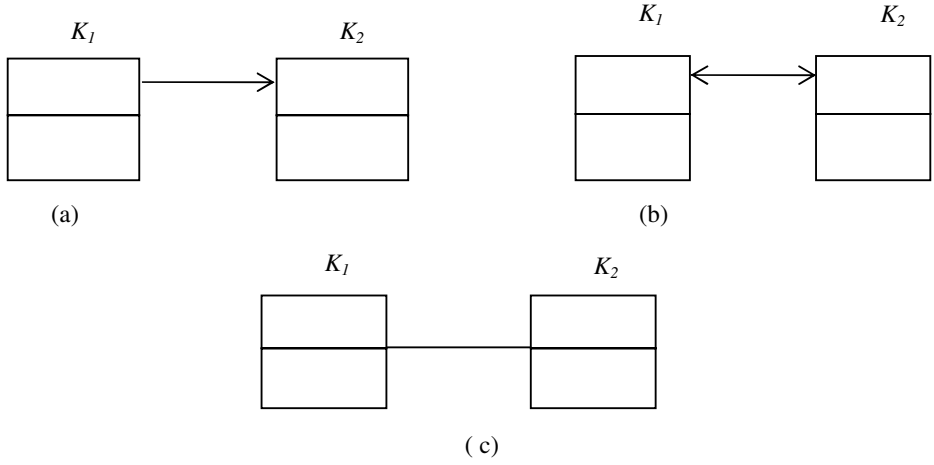



**Figure 1.10.** Knowledge systems that are operationally dependent (a), operationally connected (b), and bound by weakly defined connection (c).

Sometimes it is possible to detect that knowledge systems interact, but it is impossible or difficult to make precise the way in which they interact. Then we say that we have a *weakly defined connection of knowledge systems* and will not apply any attribute to specify how they are connected. In such a case we use a line that can connect any parts of two knowledge systems. The respective graphical notation is shown in Fig. 1.10(c).

## 1.11.5. Examples of knowledge architectures

*Program synthesis tools.* In this section, we consider knowledge systems as architectural components of knowledge-based tools. A simple example of efficient usage of connected knowledge systems is in the deductive program synthesis. There are different

methods of deductive program synthesis. Their main idea is to construct a proof that solution of a given problem exists, and thereafter to extract a program from this proof that solves the problem. A knowledge system of logic is used for constructing a proof of solvability. Another knowledge system is needed for performing computations, and this knowledge system is CCF (see Section 1.1). A code generator transforms a proof into an executable code and connects these systems. It implements a mapping C from interpretations of logical formulas into objects of CCF, as shown in Fig. 1.11.

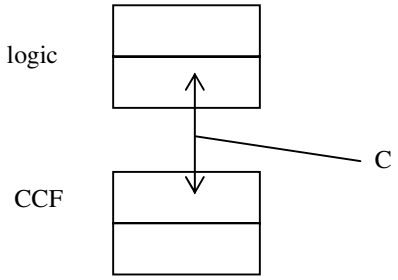



**Figure 1.11.** Knowledge architecture of deductive program synthesis software.

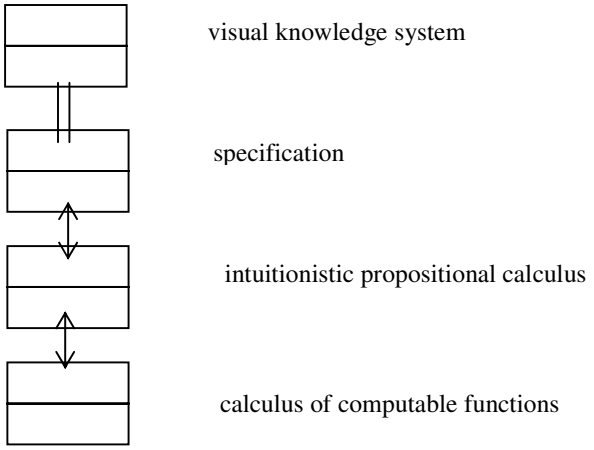



**Figure 1.12.** Calculi of CoCoViLa.

New knowledge systems have been added to a more recent software tool with program synthesis features – CoCoViLa [16]. It has a visual knowledge system for developing and using visual specifications. Meaning of a visual specification is a specification in a language similar to the specification language of computational frames that are handled by means of a specification calculus. A specification is translated into the intuitionistic propositional logic and this logic is used for deductive program synthesis. This gives us the architecture shown in Fig. 1.12.

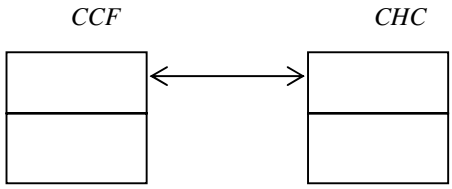



**Figure 1.13.** Knowledge architecture of Prolog.

*Logic programming tools. L*ogic programming tools include CCF and a logic knowledgel, but these knowledge systems are connected in a way that is different from program synthesis tools. A good example is Prolog. It includes calculus of Horn clauses (CHC) and the calculus of computable functions CCF. How Prolog works can be explained in terms of logic. In this case, one uses CHC as thre main knowledge system. But some calculations should be made for evaluating terms in atomic formulas and for implementing some predicates in this case as well. This means that CHC uses CCF, and we say that CHC is operationally dependent on CCF. From the other side -- Prolog can be explained in terms of computations. In this case, logic is used to help to control the computations, and we see that CCF is operationally dependent on CHC. The two knowledge systems: CCF and CHC are operationally dependent from one another, hence they are operationally connected as shown in Fig. 1.13.

## 1.11.6. Ontologies and knowledge systems

Let us assume that we have a knowledge system that we are going to fill with knowledge. We have a language for defining knowledge objects, and an interpretation that will give meaning of the objects. However, we need some initial knowledge to begin with. This knowledge constitutes an ontology that will be used for writing meaningful texts in the language of our knowledge system. By definition, *ontology* is a system of interrelated concepts that is used for presenting knowledge in some knowledge domain. Roughly speaking, an ontology is a set of concepts (names and their meanings), as well as some very basic relations between the concepts: being a subconcept (i.e. the inheritance relation), being a part of, being a property of, being a value of a property. In ontologies developed for a restricted application domain the relations may be more specific.

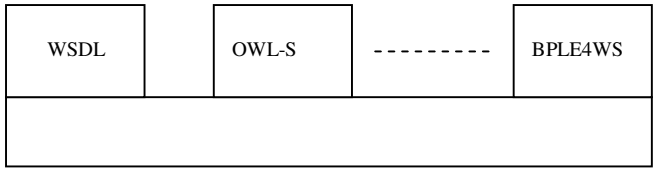



**Figure 1.14.** Knowledge systems of web services

Usage of ontologies is common in developing web services and a semantic web that is based on the knowledge about services. There are numerous knowledge systems for handling web services in a more or less intelligent way, and there are numerous ontologies for filling them with knowledge. The problem is that these knowledge systems and ontologies are difficult to put together and use jointly. Let us discuss the usage of ontologies on the example of web services in some detail.

The situation in web services is roughly as shown in Fig. 1.14. On the basic level there is a common set of meanings for the knowledge systems of web services. It comprises service presentation, detection, implementation and coordination (called orchestration and choreography). There are a number of languages (and, consequently, knowledge systems) for expressing this meaning. In Fig. 1.14 we see the knowledge systems called by the names of the languages of web services: WSDL, OWL-S, BPEL4WS etc. We can assume that they all have the same set of meanings (this can be an extended set of meanings as defined in Chapter 1.11.3). These systems cover different aspects of web service creation and application. However, they do not fit well together. There is no system that could cover all aspects of the web services. Different knowledge systems cover the same essential aspects, and do it in a slightly different ways. A problem of translating from one language (from one knowledge system) into another arises. This problem has not been solved yet. Some ontologies have been developed independently with the goal to be applicable with different web service knowledge systems. Also this approach works only to some extent. In order to support this approach, one has to be able to translate the independent ontologies in a language of a particular knowledge system.

The same problems are common for other applications of ontologies. Attempts to develop one universal ontology for all have not been successful, and different specific ontologies have been difficult to use together.

## 1.12. Summary

We have discussed several knowledge representation forms. Each of them can be formalized as a calculus. Experience shows that there is no universally efficient knowledge representation and handling technique. On the contrary - a number of very different methods have been developed for knowledge handling in different domains. When solving complex problems, one has to combine several knowledge systems.

There is a conflict between the generality of a calculus and the efficiency of derivation in it. The more specific calculus we build, the better methods we can find for making inferences in this calculus. However, we also need some means for representing general knowledge. This is a good reason for combining different calculi, i.e. different knowledge systems. Fig. 1.15 shows a classification of knowledge systems.

The first category is mainly based on logic. The positive properties of knowledge systems of this category are straightforward derivability, soundness and completeness.

The figure shows a variety of knowledge systems and relations between them. On the higher level, one can divide these systems in three categories:

- *symbolic KS*
- *rule-based KS*
- *connectionist KS.*

Among the symbolic knowledge systems we have, first of all, Post's systems as the most general form of knowledge representation and handling. Various form of logic are used for symbolic representation of knowledge. Clausal calculi and, in particular, Horn clauses are widely used due to their intuitive clarity and efficiency. In order to express specific features like nonmonotonic character of knowledge, more elaborated logical means have been developed.

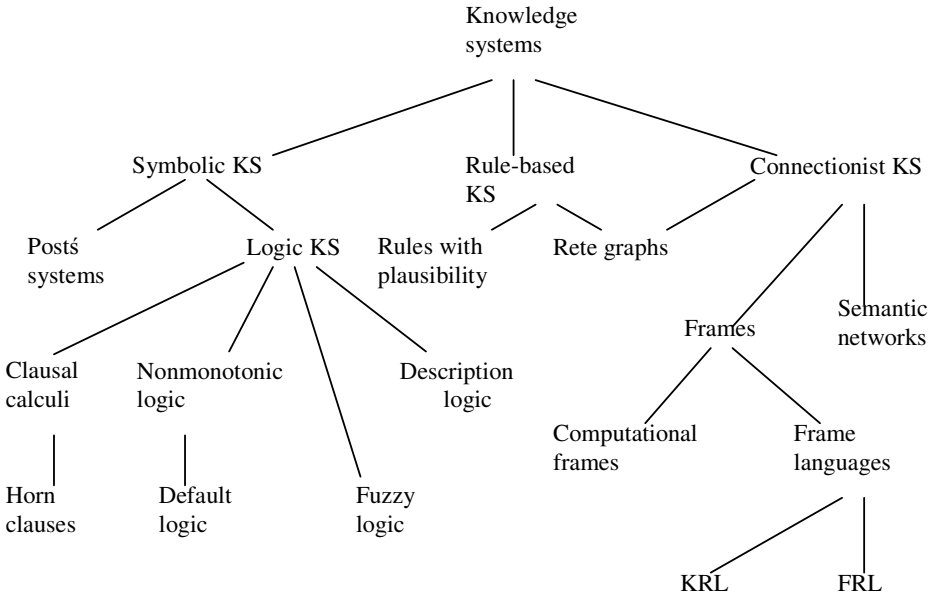



**Figure 1.15.** Classes of knowledge systems

We see in the category of symbolic KS also *description logic* and *fuzzy logic* that have not been discussed in the present chapter. Description logic has been developed in AI and is being used mainly in ontology engineering today. Fuzzy logic originates also from artificial intelligence field, it was developed by L. Zadeh in the seventies of the last century [52]. It is useful for expressing facts with some amount of vagueness, when one has to make statements with concepts like recently, not much, warm etc. In these cases, one is often unable to give precise values of time, amount of something or temperature, but has to make decisions anyway. Fuzzy logic is successfully used in control engineering.

Rule-based systems include knowledge that can be easily presented in the form of condition-action rules. The rules are implementation oriented, guaranteeing good computability. They are widely used, in particular, in expert systems for making decisions. Rules can have diffeent forms, may include plausibilities and can be composed in tables or graphs. Even decision tables and decision trees can be included in this category, although we have not shown them in Fig. 1.15. It is quite difficult to deal with formal properties, e.g. completeness and consistency, of rule-based systems. Also introducing modularity is complicated in rule-based systems.

The third category – connectionist knowledge systems rely on explicit representation of connections (relationships) between the items. Here we have semantic networks and frames. Although frames do not require explicit description of connections, the main idea of inheritance was very much connectionist in its origin, and first implementations of frames were in the connectionist style. The positive properties of the knowledge systems of this category are modularity, conceptual simplicity and explicit structure of inheritance. The frame languages FRL and KRL shown in figure have been only briefly mentioned in this chapter. They have some historic relevance as the first knowledge representation languages in the style of specification languages that have become popular later in many application domains.

There are other ways of knowledge representation not discussed in this chapter, for instance, artificial neural nets that will be discussed in Chapter 3. Knowledge in a neural net is partially encoded in a structure of the net, and partially in weights of connectons. Finally, we would like to stress that, in an abstract way, all knowledge representation and handling mechanisms can be considered uniformly as interpreted free deductive systems presented in the beginning of this chapter. This representation is useful for theoretical considerations, and for describing knowledge architecture of complex intelligent systems as shown in the end of the present chapter.

## 1.13. Exercises

1. Define a Post´s system that enables one to split any finite set with at least two elements represented by enumeration of its elements like $\{a_1, \ldots, a_n\}$ into two sets $\{a_1,\ldots, a_i\}$ and $\{a_{i+1},\ldots, a_n\}$ for any $i=1,\ldots, n-1$.

2. Define a PS that enables one to split a finite set represented by enumeration of its elements, e.g. *{a,q,b}* in an arbitrary way.

3. Develop a Post's system that creates words "*x covers y"* where *x* and *y* are simple concepts defined as follows. Simple concept is a tuple $<a_1,a_2, a_3>$, where $a_i$ $(i=1,3)$ is +, - or the wildcard *. Simple concept can be, for instance, ***, ++-, *+*, -*- etc. The Post's system should generate all words where on the right side is a simple concept that is on the left side or derived from the left side simple concept by substituting + or - instead of some *. For instance, it should be able to generate the words:
*-* *covers* +–*, *-* *covers* *-+ etc.

4. Write computational frames of concepts *Rectangle* and *Rhombus* in the specification language presented in this chapter.

5. Find the most general unifier for the pairs of literals, if a pair of literals is unifiable, and if it is not unifyable then explain why:

| | |
|---|---|
| *Color(Tweety, Yellow)* | *Color(x,y)* |
| *R(F(x),B)* | *R(z,y)* |
| *R(F(y),x)* | *R(R(x),F(B))* |
| *Loves(X,Y)* | *Loves(y,x)* |
| *Pattern(abc,x,abx)* | *Pattern(abc,y,aby)* |

6. Prove by refutation the theorem below, i.e. show that *c* is derivable from the formula $(a \wedge b \supset c) \wedge (d \supset a) \wedge (d \supset b) \wedge d$:

$(a \wedge b \supset c) \wedge (d \supset a) \wedge (d \supset b) \wedge d \vdash c$

7. Build a decision tree for classifying birds depending on their ability to fly, size, living area etc.

8. Describe rules for calculating a square root of *x* by using so called Babylonian method, knowing that it is done by applying the formula

$y' = 0.5(y+x/y)$

for finding the next approximation y´ from a given approximation y. Initial approximation can be taken equal to $0.5x$.

9. Given the facts in Prolog:

```
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
```

Find answers to the following questions and explain how they are obtained:

```
?- parent(bob,pat).
?- parent(liz,pat).
?- parent(X,liz).
?- parent(bob,X).
```

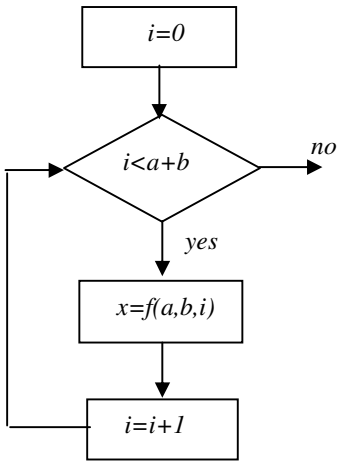
10. Define a relation in Prolog:

```
duplicate( List, List1)
```

that duplicates each element of the list L in L1.
Example:

```
duplicate([a,b,c,d,e],[a,a,b,b,c,c,d,d])
```

11. Present an algorithm given by the following program scheme in the form of a set of productions. (Hint: introduce additional Boolean variables for showing a state of computations on the scheme.)



12. Define a relation in Prolog that has the following signature:

```
duplDeep(List1,List2)
```

and duplicates each element of the list L in L1 and also in each of its sublists.
Example:

`duplDeep` ([3,[5,1]],[3,3,[5,5,1,1],[5,5,1,1]).

# 2. Search

## 2.1. Search problem

Search is a universal method of problem solving that can be applied in all cases when no other methods of problem solving are applicable. People apply search in their everyday life constantly, without paying attention to it. Very little must be known in order to apply some general search algorithm in the formal setting of the search problem. However, if additional knowledge can be exploited to guide the search, then the efficiency of search can be drastically improved. Search is present in some form almost in every AI program, and its efficiency is often critical to the performance of the whole program. A great variety of search methods have been developed, which take into account the specific knowledge about particular search problems (see, for instance [40]).

A *search problem* in the most general form is a description of the condition that must be satisfied by its solution, and a way to find candidates for the solution. The latter, in its turn, can be given in different forms. In the algorithms of this chapter, we shall use the predicate

*good(x)*

in order to designate the condition for the solution. For generating the objects that are candidates to be tested, we can use the functions

*first()* – for finding an object when no other objects have been given
*next(x)* – a *generator function* for finding a new object, when an object $x$ is known.

This gives us already the possibility to build a simple *search algorithm with a generator function* A.2.1.

**A.2.1:**

```
x=first();
until good(x)
do
    x=next(x)
od
```

Another way to represent a search problem is to give the predicate *good(x)* and, besides that, to give the set of testable objects as a whole, leaving the development of the search operators to the designer of algorithms. The set of testable objects is called *search space*. Its properties determine the suitability of search algorithms. If a search space has a finite number of elements, then one can use a simple *search algorithm on search space* that is as follows:

**A.2.2:**

```
for x ∈ searchSpace
        do
                if good(x) then success(x) fi
        od;
failure()
```

To improve the search, one can use a set called *options* that contains only the objects that may be interesting to test at a certain step of the search instead of the whole search space. This set will be modified during the search by the procedure *modify(options).* The most important function in the search algorithm will be a *selector function* that we denote by *selectFrom(options)* which performs the selection of a new object from *options* to test at each search step. Other building blocks of algorithms will be a predicate *empty(x)* for testing the emptiness of a set *x,* and a procedure *initialize(options)* for determining the initial content of *options*. A *search algorithm with selector function* that uses these building blocks has the following general form:

**A.2.3:**

```
initialize(options);
while not empty(options)
do
        x=selectFrom(options);
        if good(x) then success(x) else modify(options) fi
od;
failure()
```

To be able to build suitable search functions: *selectFrom()* and *modify()*, one must know as much as possible about the search space. We call these functions also *search operators,* because they determine the actual search process. Instead of *initialize(options)* one can always use the assignment *options=searchSpace,* although this is the least precise action to be taken for initialization. Sometimes the search space is a precisely described mathematical object like Eucleidean space, for instance. One can hope to be able to apply well-developed algorithms then, like numerical algorithms for solving equations, or optimization algorithms, depending on what the function *good(x)* in the particular problem is. We shall not consider these algorithms here. However, it is useful to remember that these algorithms are just specific search algorithms that have been developed with the good knowledge of particular search problems.

We can define the search space in a slightly different way. It can be a set of search states, where a *search state* is an object to be tested, possibly, together with some additional information obtained during the search. In AI, such a search space is often represented as a graph with the nodes representing the states of search, and arcs leading from a state to all states that can be immediately reached from this state. This is a *search graph.* If the information associated with each search state determines also the way in which the state was obtained, the search graph becomes a tree. This is a *search tree.*

Search in a search space is performed by starting from some initial search state, and then proceeding from state to state. There is a modified search problem at each of the search states passed by the search process. This is the search problem on the part of the search space not yet explored, possibly, with additional knowledge obtained during the search. This observation allows us to define a search space also as a set of search problems generated by modifying the initial problem by means of the search operators.

There is an interesting class of search spaces called and-or trees with states as search problems that appear often in the search related to games. An *and-or tree* is a tree where each node represents a problem. Besides that, each node is either an and-node or an or-node. For solving the problem of an and-node one has to solve problems of all its immediate descendants. For solving the problem of an or-node one has to solve the problem of one of its descendant. Problems of the terminal nodes (leaves) of an and-or tree are primitive, i.e. either directly solvable or obviously unsolvable.

## 2.2. Exhaustive search methods

### 2.2.1. Breadth-first search

We start the presentation of search algorithms with the algorithms of systematic search through the whole search space. These algorithms are called also *brute force search algorithms*, because they are intended for exhaustive search under conditions where little is known about a particular search problem and the problem is so simple that it is possible to solve it by applying brute force with little intelligence.

The idea of the *breadth-first search* can be explained on an example of a simple program. It uses the following functions for moving on a given search tree from a node *p* to the next node:

*next(p)* – the node next to *p* on the same layer as *p* is in the tree;
*down(p)* – the first node on the level next to the layer of *p* in the tree;
*empty(p)* – true, if the node *p* is empty (absent) in the tree.

The algorithm takes the root of a search tree as the argument *p.* The *breadth-first search algorithm* is as follows:

**A.2.4:**

```
bf(p) = while not empty(p) do
                    while not empty(p)
                    do
                            if good(p) then success(p) fi;
                            p=next(p)
                    od;
                    p=down(p)
            od;
            failure()
```

This program searches through a tree in a regular way, layer by layer, starting from its root and going down to the next layer only when the previous layer has been completely searched. This guarantees us that, given enough space and time, the algorithm will find a solution of the search problem even in the case of an infinite search tree, if the solution exists.

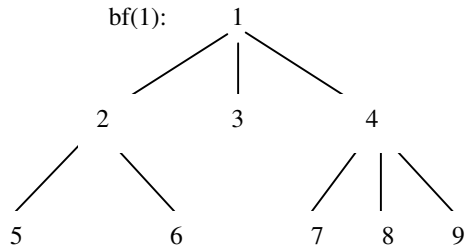



**Figure 2.1.** Breadth-first search order

Fig.2.1 shows the order, in which the nodes of a particular search tree are visited by this program. A difficulty with this program is that the functions *next()* and *down()* are not common functions for trees, and they can not be programmed efficiently – one has to remember all unvisited nodes of the layer where the search is performed, as well as the detected nodes of the next layer, if no extra information is available.

If we compare the algorithm A.2.4 with the algorithm A.1.1 for brute force search of a word generated by a Post's system, we can see that the latter is in essence a breadth-first search algorithm. The sets $S_1$, $S_2$,… correspond to the layers of a search tree, and these sets include the information describing a current layer, hence the operators *next()* and *down()* were easy to construct in that case.

To obtain a slightly better performance of the breadth-first search algorithm, we introduce now a very useful data structure that will be used also in many other algorithms. This is an *open-list* of objects to be tested in some order. In this particular case it is constituted of the nodes of the current level of the search tree which have not yet been tested, and of the nodes of the next level which can be accessed from the already tested nodes. We use the following notations:

*open* – list of objects to be tested;
*empty(open)* – true when *open* is empty;
*first(open)* – the first node of the list *open;*
*rest(open)* – the list *open* without its first node;
*succ(x)* – successors of *x* in the tree;
*(L,L1)* – concatenation of lists, i.e. the list obtained from the list *L* by adding all elements of the list *L1* to its end.

The algorithm *bfO(p)* of the *breadth-first search with open-list* that begins the search from the given root *p* of a search tree can be represented now as follows:

**A.2.5:**

```
bfO(p)= open=(p),
        while not empty(open)
        do
                x=first(open);
                if good(x) then
                        success(x)
                else
                        open=(rest(open),succ(x))
                fi
        od;
        failure()
```

The loop invariant here is the condition that all predecessors of the nodes from *open* have been already tested, the nodes yet to be tested still are either in *open* or they can be reached from these nodes in the tree. Actually, the list *open* represents a border between the tested and untested parts of the tree, and it moves to the leaves of the tree starting from its root. The space complexity of the algorithm *bfO(p)* is determined by the size of the list *open.* This size depends on the shape of the search tree, and equals to the maximal number of nodes on one level. If the branching factor (average number of descendants of a node) of the tree is *d* and its height (maximal length of a path) is *n*, then the maximal number of

nodes on one level of the tree is $d^{n-1}$. The space complexity of the algorithm *bfO(p)* is exponential with respect to the depth of search: $O(d^{n-1})$.

Because this is an exhaustive search algorithm, its time complexity is linear with respect to the size of the search space.

## 2.2.2. Depth-first search

Depth-first search algorithm is another brute-force algorithm, but it can be adjusted to a particular search problem in several ways. We explain the *depth-first search* algorithm in terms of the following notations that apply to a fixed search tree:

s*ucc(p )*-- successors of the node *p;*
*empty(p)* – true when *p* is empty.

The *algorithm of depth-first search* is represented by the recursive function *df(p)* that takes the root *p* of a nonempty search tree as an argument:

**A.2.6:**

```
df(p) = if not empty(p) then
                if good(p) then
                        success(p)
                else
                        for q∈ succ(p) do df(q) od
                fi
        fi
```

This algorithm is rather economical with space – it stores only information along the path from the root of a tree to the current node, i.e. its space complexity is

*O(log(k))*

with respect to the size *k* of the search space, or

*O(d)*

with respect to the depth *d* of search in a search tree.

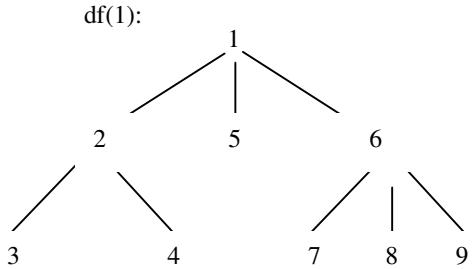



**Figure 2.2.** Depth-first search order

Fig. 2.2 shows the order in which nodes of a particular search tree are visited by this algorithm. Compared to the breadth-first search, this algorithm has the following disadvantage. It cannot be applied in the case of an infinite search tree, because it may proceed into depth along a branch of a search tree infinitely, if the solution is on another branch, then it will not be found. In order to overcome this difficulty, we must restrict the depth of search *d*.

To be still able to perform exhaustive search in a depth-first way, we must iterate the search by gradually increasing maximal depth of search. These improvements give us the *iteratively deepening depth-first search* algorithm represented by two programs *idf(p)* and *dfr(p,d)*. The first program *idf(p)* just gives the maximal search depth to the program *dfr(p,d)* that performs depth-first search with restricted depth. In this algorithm, *initDepth* is the initial search depth, *incrd* is increment of search depth for a new iteration of the *dfr(p,d)*. The function *success(p)* sets *found* = *true* for *idf(p)*.

**A.2.7:**

```
idf(p):
        d=initd;
        found=false;
        while not found & d < dmax do
                dfr(p,d);
                d=d+incrd
        od;
```

where the program *dfr(p,d)* is as follows:

```
dfr(p,d) = while d > 0 & not empty(p) do
                if good(p) then success(p)
                else for q∈ succ(p) do dfr(q,d-1) od
        od
```

The drawback of this algorithm is that some search is repeated, when the depth *d* has been increased. However, this increase is not too big – consider the ratio of leaves and all nodes of a tree with branching factor *k*. Even for a binary tree where the branching factor is 2 this ratio is 1:2.

## 2.2.3. Search on binary trees

*Binary trees*, i.e. the trees where each node has at most two descendants, are especially convenient structures for data representation, and they are widely used. Let us introduce the following two functions on a binary tree:

*left(p)* – the left descendant of the node *p*
*right(p)* – the right descendant of the node *p*.

These two functions suffice for representing various search algorithms on binary trees. Two different (but symmetrical with respect to each other) algorithms of *depth-first search on binary trees* are the following:

**A.2.8:**

```
ulr(p)= if not empty(p) then
            if good(p) then
                success(p)
            else ulr(left(p));
                 ulr(right(p));
            fi
        fi
```

**A.2.9:**

```
url(p)= if not empty(p) then
            if good(p) then
                success(p)
            else url(right(p));
                 url(left(p));
            fi
        fi
```

These algorithms are written in a really economical way, however, they do not show the failure, when a solution does not exist. This has to be detected after performing the algorithm as the absence of success (result).

Any tree can be represented as a binary tree. Let us have a tree with the functions *next(p)* and *down(p)* defined as in the previous section. If we rename these functions and call them respectively *right(p)* and *left(p),* we get immediately a possibility to apply the algorithms *ulr(p)* and *url(p)* for arbitrary trees, viewing them as binary trees. Let us notice that in this case the algorithm *ulr(p)* is just a recursive representation of the depth-first exhaustive search algorithm for the tree represented as a binary tree. We have used the names *ulr* and *url* as acronymes for showing the order of testing the neighboring nodes: upper-left-right and upper-right-left. There are four other obvious possibilities to perform the search on binary trees, which can be called: *lru, rlu, lur* and *rul*.

## 2.3. Heuristic search methods

Performance of the search can be drastically improved by using specific knowledge about a search problem to guide the decisions made where to search. This knowledge can be presented in the form of *heuristics* – rules that guide the decision-making during the search. This gives us the class of *heuristic search algorithms*. It is quite obvious that heuristics will depend very much on the class of solvable problems. Still, there are methods of heuristic search that are widely applicable after some concretization of search functions. These methods will be presented in this chapter in the form of algorithms.

### 2.3.1. Best-first search

The best-first search algorithm is a simple and general heuristic search algorithm. We get it by improving the depth-first search algorithm in the following way. We change the function *down(p)* so that it will not take simply the first node of a search tree one level down, but will choose the node to be known as a good candidate for the best choice. Let us call the new function *bestDown(p)*. We need also a function *succ(p)* that provides the set

of not yet attended successors of *p*. Then we get a simple recursive version of the *best-first search* algorithm:

**A.2.10:**

```
BestFirst(p) =
        if good(p) then success(p) fi;
        while not empty(succ(p)) do
                BestFirst(bestDown(p))
        od;
```

There is the following difficulty with this algorithm. The function *bestDown(p)* must exclude a tested node from the set of unattended nodes. It is difficult to do in the present version of the algorithm. Better best-first search algorithms can be developed on the basis of the beam search that we are going to discuss in the next section.

## 2.3.2. Beam search

*Beam search* is a search method where almost the whole search process can be determined by heuristics. The search is performed in the subset of the search space called the *beam*. This set contains the most promising known candidates for solution. Initial value of the beam can be any element of the search space from which the solution is accessible by means of the search operators. The beam is updated at every search step on the basis of the knowledge obtained during the search. The beam is always a finite set, even when the search space is infinite. The beam is analogous to the set *open* in the breadth-first search algorithm, except that the beam need not be a complete border between the attended and unattended nodes. The beam is not only extended, but also decreased, using heuristics, if it grows too large. This makes the beam search incomplete in general – solutions may be lost when the beam is decreased. The following notations are needed for the beam search algorithm:

*open* – the set of nodes to be tested currently (the *beam*);
*initstate* – element of the search space where to start the search;
*succstates(x)* – expands the set *x* (the beam) by adding to it all untested successors of its elements;
*score(x)* – evaluates the elements of the set *x* and assigns them the estimates of their fitness;
*prune(x)* – prunes the set *x* by dropping the elements with lower estimated fitness.

We shall use the predicate *good()* with the parameter *open* here showing that it depends on the contents of this set. Determining whether *good(open)* is true, i.e. whether a solution is in the set *open* is not a trivial task – it takes time. Sometimes this task can be better performed in operators *succstates()* and *score()* where one has to handle every element of the *open* set anyway. For simplicity of the presentation we are not showing this in the present algorithm. The most general *beam search* algorithm is the following:

**A.2.11:**

```
open={initstate};
while not empty(open) do
```

```
        if good(open) then success() fi;
        candidates=succstates(open);
        score(candidates);
        open=prune(candidates)
od;
failure()
```

The function *succstates(open)* depends on the search process, because it extends the set *open* only with unattended elements of the search space. To simplify this function, we can introduce an additional set called *closed* that consists of all already attended elements of the search space. This gives us the following *improved beam search* algorithm:

**A.2.12:**

```
open={initstate};
closed={};
while not empty(open)do
        if good(open) then success() fi;
        closed=closed ∪ open;
        candidates=succstate(open)\closed;
        score(candidates);
        open=prune(candidates);
od;
failure()
```

This algorithm can be taken as a basis for the improved algorithm that tests the best candidate at each step and uses a heuristic function *bestFrom(open)* to do this. It is another *best-first search* algorithm:

**A.2.13:**

```
open={initstate};
closed={};
while not empty(open) do
        x=bestFrom(open);
        if good(x) then success(x) fi;
        closed=closed ∪ {x};
        candidates=open ∪ succ(x)\closed;
        score(candidates);
        open=prune(candidates)
od;
failure()
```

It is easy to see that the speed of the best-first search algorithm A.2.13 depends on the quality of the function *bestFrom(open).* Another important function is *prune(candidates).* This function should not throw away candidates that may lead to the solution.

## 2.3.3. Hill-climbing

If we constrain the beam search algorithm in such a way that the beam contains precisely one element, we get the *hill-climbing algorithm* that is a popular heuristic search algorithm. In this case we can use, instead of the two functions *score()* and *prune()*, the

function *bestFrom(candidates)* that gives the best element from *candidates* or an empty result, if the set *candidates* becomes empty. In this case the search operator *bestFrom()* must find the actual best continuation at every step of the search path. This is possible, for instance, if gradient of the fitness function can be computed at every step.

**A.2.14:**

```
x=initstate;
closed={};
while not empty(x)do
        if good(x) then success(x) fi;
        closed=closed ∪{x};
        candidates=succ(x)\closed;
        x=bestFrom(candidates)
od;
failure()
```

Originally, the hill-climbing algorithm did not use even the set *closed*, and had the following very simple description:

**A.2.15:**

```
x=initstate;
while not empty(x)do
        if good(x) then success(x) fi;
        candidates=succ(x);
        x=bestFrom(candidates)
od;
failure()
```

This algorithm is very economical with respect to space – it stores only one element, and it is good for searching for the best element in a search space where the fitness function of elements changes monotonously along a search path – a requirement that is fulfilled for the convex optimization problems, for instance. In this case, one is certain that there exists only one local optimum of the fitness function, and this is also its global optimum.

## 2.3.4. Constrained hill-climbing

Practical problems of optimization have often constraints on the search space, e.g. inequalities that cut off a part of an Eucleidian space where the search is performed. One can use a given set of forbidden objects – *forbidden* for presenting the forbidden part of search space. Then we get *constrained hill-climbing* algorithm A.2.16.

**A.2.16:**

```
x=initstate;
closed={forbidden};
while not empty(x) do
        if good(x) then success(x) fi;
        closed=closed∪{x};
        candidates:=succ(x) \ closed;
        x=BestFrom(candidates)
```

***od;***
*failure()*

This seemingly small difference can cause considerable difficulties, especially, if the search is otherwise performed in a good space, e.g. Eucleidean space. In particular, optimization problems with convex fitness function that in principle have one optimum may become multimodal optimisation problems when constrained by a *forbidden* set.

## 2.3.5. Search with backtracking

Backtracking is a way to apply the trial-and-error paradigm in problem solving [5]. It says that one can try to solve a problem by trying to build a solution gradually, step by step. When it becomes obvious that a partial solution is not correct, one must throw away it, or part of it, and continue the search for solution.

*Search by backtracking* can be explained in a very simple way by factorizing the search space into several subspaces $D_1,D_2,...,D_n$, so that the whole search space will be their product $D_1 \times D_2 \times .. \times D_n$. A problem is to find a tuple $x=(a_1,a_2,...,a_n)$ that satisfies the predicate *good(x)*, and $a_1 \in D_1$, $a_2 \in D_2,...,a_n \in D_n$. The predicate *good()* in this setting is extended so that it can be applied also to partial solutions $(a_1,a_2,...,a_k)$, where $k<n$.

The search starts by selecting a candidate $a_1$ in the subspace $D_1$. Thereafter, the candidate $a_2$ in the subspace $D_2$ is selected and a partial solution $(a_1,a_2)$ is formed etc. This process is continued until the whole solution is found, or it becomes clear that a partial solution, let us say, $(a_1,a_2,...,a_k)$, $k<n$ is unsatisfactory. If the latter happens, the last component $a_k$ of the partial solution will be dropped and, if there still are unchecked elements in $D_k$, a new candidate from $D_k$ is selected. This is a *backtracking step.* If there are no more candidates to select from $D_k$, one more backtracking step is done and the partial solution is reduced to $(a_1,a_2,...,a_{k-2})$. The search continues until a solution is found, or backtracking steps have reduced the partial solution to the empty tuple and, at the same time, there is no more possibility of choice in $D_1$. The *backtracking algorithm* A.2.17 uses the following notations:

The function call *F((), D, D)* where $D=( D_1,D_2,...,D_n)$ finds the answer $(a_1,a_2,...,a_n)$ if such a tuple exists in $D_1 \times D_2 \times .. \times D_n$;

$V_1, \ldots, V_n$ are the sets of untested alternatives in $D_1,D_2,...,D_n$ respectively;

*accept (x, . . . , y)* is true, if *x, . . . , y* is an acceptable combination of elements from $D_1,D_2,...,D_i$;

*selectFrom (x)* produces an element of *x,* if *x* is nonempty, otherwise produces *nil*.

**A.2.17:**

*F*$((a_1, \ldots, a_i), (V_1, \ldots, V_n), D)$ =
  ***if*** $i == n$ ***then***
    $(a_1, \ldots, a_n)$
  ***fi;***
  *L:* ***while not*** *empty*$(V_{i+1})$ ***do***
    *x=selectFrom* $(V_{i+1})$;
    ***if*** *accept* $(a_1, \ldots, a_i, x)$ ***then***
      *F*$(( a_1,\ldots , a_i, x),(V_1,\ldots,V_i,V_{i+1}\backslash\{x\},D_{i+2},\ldots$
      $\ldots,D_n),(D_1,\ldots,D_n))$;

```
            break L
        fi
    od
    if i==0 and empty(V_{i+1}) then
        failure()
    elseif empty(V_{i+1}) then
        F(( a_1,... , a_{i-1}),(V_1,...,V_{i-1},D_i,D_{i+1},...
            ...,D_n),(D_1,...,D_n))
    fi
```

## 2.3.6. Search on and-or trees

Another, and a slightly more complicated backtracking search algorithm is the search on an and-or tree that has interleaved layers of or-nodes and and-nodes. Search is successful in an and-node if and only if it is successful in all its immediate descendants. It is successful in an or-node if and only if it is successful in some of its immediate descendants. Success in any terminal node is determined by some external rule. We shall use the following functions in the *and-or search algorithm*:

*succ(p)* - set of immediate descendants of the node *p;*
*terminal(p)* - true iff *p i*s a terminal node;
*good(p)* - true iff the terminal node *p* is good (this is an external decision procedure).

The following algorithm produces a subtree of an and-or tree given by its root *p*. The result of a search must be a tree, because it must show a good solution for every successor of any and-node included in the result. This tree is empty, if the search is not successful, otherwise it is the detailed result of successful search, i.e. a tree of successful nodes represented in a list notation, e.g. *(a(b(c),d))* for a tree where the root *a* has successors *b* and *d*, and the node *b* has a successor *c*.

The following algorithm searches on trees that have an or-node as a root.

**A.2.18:**

```
and_or(p)=
    L: for x∈ succ(p) do
        if terminal(x) and good(x) then
                return x
        elif terminal(x) and not good(x) then
                continue L
        else
            result1= ();
            for y∈ succ(x) do
                if empty(and_or(y)) then
                        continue L
                else
                        result1=append(result1,and_or(y))
                fi
            od;
```

```
                return (x(result1))
            fi
        od
```

The algorithm *and_or(p)* works as follows. The loop *L* searches among immediate descendants *x* of the root *p* for a good terminal node or for a subtree that presents a solution. The inner loop tests all successors *y* of the current node *x*. As *x* is on the layer of and-nodes, this node is good only if all subtrees with roots *y* as its immediate descendants contain a solution. This is checked by recursive call of the algorithm *and_or(y)*.

This algorithm is used in games of two players. An or-node is for selecting a move of a player and an and-node is for evaluating all possible moves of its adversary. For a winning strategy, all moves of the adversary should be answered by winning moves of the player, because the adversary is free to select any of them.

## 2.3.7. Search with dependency-directed backtracing

A backtracking algorithm can be improved in the following way. When a partial solution is found to be unsatisfactory, the reason for this can be analyzed, and the place of

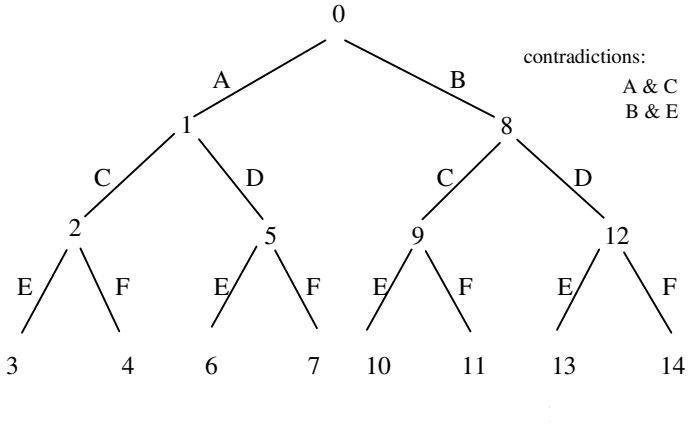



**Figure 2.3.** Dependency-directed backtracking

the bad choice that caused the fault can be determined. This allows making longer jumps at backtracking steps and spares the effort of search. It is also possible to memorize the unsatisfactory decisions in order to avoid their usage when the search is continued. This gives us *dependency-directed backtracking*. Fig. 2.3 shows a search tree where the decisions are shown on its arcs and the following conjunctions describe conditions of bad decisions – contradictions:

*A&C* and *B&E*.

When looking for all possible solutions, the dependency-directed backtracking gives the following sequence of states:

0 1 2 3 5 6 7 8 9 10 11 12 14.

Backtracking to appropriate choice avoids the state 4, even if the contradiction *A&C* is detected at the state 3. Memorizing contradictions met during the search avoids the state 13.

## 2.3.8. Branch-and-bound search

*Branch-and-bound search* is a method of solving optimization problems. It is being used in areas like operations research and combinatorial mathematics [19], [22]. It can be considered as a generalization of heuristic methods like alpha-beta pruning (see below). The problem it solves can be formulated as follows.

Given a discrete search space $X$ and a *fitness function f* on $X$, find an element $x_m$ of $X$ such that $f(x_m)=max\{f(x) : x \in X\}$.

The idea of this method is to split the entire set $X$ into smaller parts and to use additional knowledge about the problem for excluding as many of these parts as possible from search, when searching for the solution. This exclusion is done by comparing upper and lower bounds of the function $f$ on the parts of $X$. Let us denote by $f^+(Y)$ the upper bound and by $f^-(Y)$ the lower bound of $f(x)$ on $Y$, $Y \subset X$. We say that $Y$ dominates $Z$, where $Y \subset X$ and $Z \subset X$, iff $f^+(Z) \leq f^-(Y)$. Obviously, if $Y$ dominates $Z$, then it is sufficient to search for the solution in $Y$, excluding the set $Z$ from the consideration. To find upper and lower bounds, one splits the sets several times, so that the sets become small enough to be estimated. The important operators of the branch-and-bound algorithm are the following:

*split(A)* – splits the set *A* into a number of smaller sets;
*M* – a set of subsets of *X* that that should include the solution;
*selectFrom(M)* – selects a set to be split from the set of sets *M;*
*prune(M)* – finds the dominated sets in *M* and prunes them out;
*card(M)* – gives maximum cardinalities of sets in *M*.

These operators are very problem-dependent and cannot be explained in any detail in general. The *alpha-beta pruning* is a concretization of the present algorithm. It presents examples of these operators. Selecting and splitting is done in alpha-beta pruning according to the rules of a game. The values *alpha* and *beta* are being used for determining the dominance and pruning. This will be demonstrated in the next section. A general schema of the branch-and-bound algorithm is the following:

**A.2.19:**

```
M={X};
while card(M) > 1 do
        A=selectFrom(M);
        M=M\A ∪ split(A);
        M=prune(M)
od
```

The operator *prune(M)* selects the element *T* of *M* that has the highest lower bound. Thereafter it prunes away all elements of *M* that have upper bound less than lower bound of *T*. It uses the notations

*lb(X)* – lower bound of elements in X;

*ub(X)* – upper bound of elements in X.

```
prune(M)=
        T = selectFrom(M);
        for X ∈ M do
                if lb(X) > lb(T) then
                        T = X
                fi
        od;
        for X ∈ M do
                if lb(T) > ub(X) then
                        M = M\{X}
                fi
        od
```

A well-known and simple branch-and-bound algorithm is the *binary search algorithm* that searches an element in an ordered finite set. This algorithm divides a set into two approximately equal parts and compares the upper bound of the part containing smaller elements of the set with the given value. This upper bound is easy to find – it is the value of the last element in the part. Depending on the result of the comparison, one or another part of the set is selected for the further search, or the search succeeds, if the compared values are equal. The algorithm is especially simple, if the set is represented as a binary tree. The following algorithm searches in a binary tree that is ordered from left to right in the increasing order. It uses the functions *left(p), right(p)* and *val(p)* for selecting the left or right subtree and taking the value of the node *p* (of the root of a tree).

**A.2.20:**

```
binSearch(x,p) =
        if empty(p) then failure()
                elif val(p) == x then success()
                elif val(p) < x then binSearch(x,right(p))
                else binSearch(x,left(p))
        fi
```

## 2.3.9. Stochastic branch and bound search

It may be difficult to estimate the maximal and minimal values of a fitness function *f* on a set. One can use the following idea of *stochastic branch and bound search*: instead of a deterministic search, perform a stochastic search with a distribution of probability of testing that is higher for the areas of the search space where the found values of *f* are better. This gives us a good algorithm for solving nonlinear multimodal optimization problems (see Gergel, Strongin [12]). The following notations are needed for this algorithm:

*distr* - probability distribution for selecting a point in the search space;

*random(distr)* - generator of random points with distribution *distr* in the search space;

*modify(distr,x,v)*- procedure of adjustment of the distribution that increases the probability of selecting a point with higher value of fitness function, taking into account the value *v* of the fitness function at the point *x*.

**A.2.21:**

```
distr= even;
        x= random(distr);
        while not good(x) do
                x= random(distr);
                distr= modify(distr,x,f(x))
        od
```

Looking closer at the operator *modify()* one discovers that, for a good performance, it requires knowledge of values of the function *f* at many tested points, because only using this knowledge, one can make a decision about the relative usefulness of a particular area of search. This knowledge must be updated and kept in memory during the search. This makes difficult the implementation of the algorithm of stochastic branch and bound search.

## 2.3.10. Minimax search

Let us consider a game with two players who make moves one after another until a final state is reached in the game. Then it is decided who is the winner and how much this player wins. Another player – the looser looses the same amount. This is called a *zero-sum game*. The player who makes the first move will be called Max, and he/she tries to maximize the result. At the same time, the other player, let us call him Min, tries to minimize the result of the player Max. It is assumed also that both players have the complete information about the game, and can make the best possible decision on this basis. Search for the best move in this case is called *minimax search*, and the respective algorithm A.2.22 performs search until the given depth is reached. This algorithm uses the following functions:

*evaluate()* – produces the resulting value for end-positions of the game, otherwise produces an estimate of the outcome of the game;

*generateLegalMoves()* – generates all possible moves in a given position;

*makeMove(x)* – makes the move *x* and, as a side effect, changes the position respectively;

*endPosition()* – true, iff the reached position is an end position of the game;

*unmakeMove(x)* – restores the position that was before the move *x;*

*saveMove(x)* – this procedure must save, as a side effect, the move *x* that corresponds to the best outcome *alpha;*

*unfavorableResult* – a constant that is less than any real outcome of the game.

The function *miniMax(depth)* returns an estimate of the best reachable result for the player Max calculated by investigating the game to the *depth* by exhaustive search and then calculating estimates of the positions. The function *miniMax()* is used recursively for evaluating the moves of the player Min as well. For this purpose, the sign of outcome is minus and *depth* is decreased by one.

**A.2.22:**

```
miniMax(depth)=
        if depth == 0 then
                return evaluate()
```

```
    fi;
    moves = generateLegalMoves();
    alpha = unfavorableResult;
    for x∈ moves do
        makeMove(x);
        if endPosition() or depth==0 then
            val = evaluate()
        else
            val = - miniMax(depth - 1)
        fi;
        unmakeMove(x);
        if val > alpha then
            alpha = val;
            saveMove()
        fi
    od;
    return alpha;
```

## 2.3.11. Alpha-beta pruning

The algorithm of *alpha-beta pruning* is for heuristic search of best moves on game trees in the games of two players, when both players apply the best tactics, i.e. they do as good moves as possible on the basis of the available knowledge. This algorithm gives the same result as the minimax algorithm for the games of two players, i.e. it finds the best moves, but requires less time, if one can use heuristics that make the choice of better moves more likely [2]. This algorithm lies in the ground of many game programs for chess and other games.

Let us consider an and-or tree shown in Fig.2.4 that describes a game between the players where one of them tries to maximize the result and another tries to minimize it. The squares represent positions where the maximizing player has a move, and circles represent the positions with moves of the minimizing player and end positions. (In general, players may have the choice of more than two moves, but this is inessential for our considerations.) Each path from the root of the tree to a leaf (terminal node) represents a particular game. The leaves are end positions, and they contain numbers that represent the results of the games. Positions reached in a game are denoted by *P1, P2*, etc.

We assume that the moves are denoted by *1* and *2* and a move *1* leads to the left when a move *2* leads to the right. For example, the game with the moves *1,1,2* goes through the positions *P,P1,P11,P112* and it ends with the result 1. If the sequence of moves is *1,1,1* then the positions are *P,P1,P11,P111* an the result is 4.

At the position *P11*, the maximizing player has a move and he sees the positions following his possible moves. So he certainly chooses the move *1* that leads to the result 4. This is known to both players and neither of them will expect that the move *2* would be done in the position *P11*. This gives the estimate 4 for the position *P11*. Analogically, the position *P12* gets the estimate 8. Now we can repeat the similar argumentation for the position *P1* where the minimizing player has to move. Obviously, he/she will choose the move *1* which gives the result 4 and not the move *2* which would give the result 8. So the estimate of the position *P1* is 4. This argumentation can be continued in the bottom-up manner up to the root of the tree, and estimates of all nodes can be found as the result of the complete analysis of the game. The estimates are shown in the Fig.2.4. We can

conclude on the basis of this analysis that the actual game that will be played is *1,1,1* with the positions *P,P1,P11,P111* and the result will be 4. But it is important that, due to alpha-beta pruning, this conclusion can be reached without testing the positions *P122, P22, P221* and *P222*.

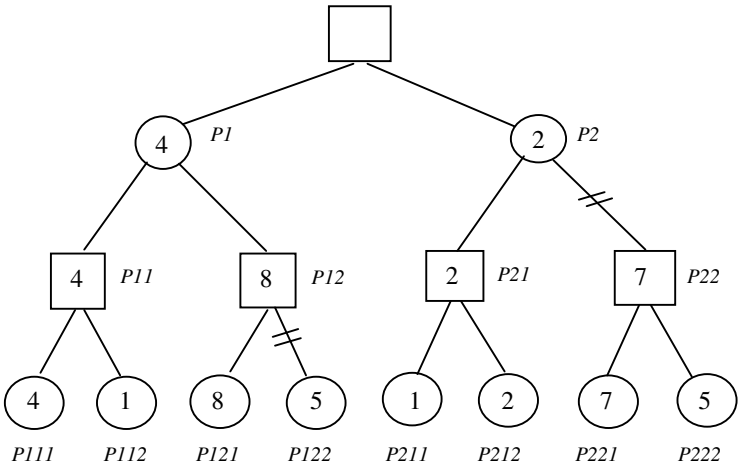



**Figure 2.4.** Example of alpha-beta pruning

Alpha-beta pruning enables us to find the best move at each position without the exhaustive search on the game tree. Remembering the best found estimates of the moves, one can avoid looking through a part of the tree which can not give better results than already known. In the present game, we can drop the analysis of the position *P2* after finding the estimate 2 for *P21*, because the estimate for *P2* can not be more than 2 now, because the minimizing player decides there, and the maximizing player will not be interested in making the move 2 from the position *P2*. The important knowledge applied here is the estimate of the best result that can be achieved for sure on the basis of search already done. For the maximizing player it is the minimal value *alpha* of the end result that can be obtained for certain. For the minimizing player it is the maximal value *beta* of the end result that, for certain, need not be exceeded. Calculation of values of *alpha* and *beta* by the alpha-beta pruning algorithm is represented by a quite simple recursive function presented as A.2.23. To evaluate moves of the minimizing player, the function *alphabeta* is called recursively. Let us notice that in the recursive call of this function the positions of parameters *alpha* and *beta* are exchanged and the sign is minus.

**A.2.23:**

```
alphabeta(p,alpha,beta) =
        if empty(succ(p)) then
                return f(p)
        else   m=alpha;
               for x∈ succ(p) do
                   t=-alphabeta(x,-beta,-m);
                   if t>m then m=t fi;
```

```
            od
            if m≥ beta then return m  else return beta fi
        fi
```

This elegant algorithm was developed by G. Adelson-Velsky in the sixties of the last century. A description of this algorithm became well known after its publication in the journal "Artificial Itelligence" [2], and since then, numerous improvements for special cases and thorough analyses of this algorithm have been published. For large game trees, like in chess, this algorithm is applied with a restricted depth of search, and approximate estimates are computed for leaves that appear as results of cutoff.

At a closer look, one can see that the problem solved by alpha-beta search algorithm is the same as the minimax problem. A version of alpha-beta search algorithm can be obtained from the minimax search algorithm by introducing minor changes. This algorithm is denoted as A.2.24, and it uses the function *makeMove()* in a more elaborated way, taking into account the values of *alpha* and *beta*, and makes a move only if these values show that this is reasonable to do. The differences with the minimax algorithm are small. The function *alphaBeta()* has parameters for the node to start search from, for search depth and for values of *alpha* and *beta,* and the algorithm has an additional statement for handling *beta*.

**A.2.24:**

```
alphaBeta(p,depth,alpha,beta) =
        if depth == 0 or terminal(p) then return evaluate() fi;
        for x∈ moves do
                makeMove(x);
                val = - alphaBeta(x,depth - 1, -beta, -alpha);
                unmakeMove(x);
                if (val ≥ beta)then
                        return beta
                fi;
                if val > alpha then
                        alpha = val;
                        saveMove(x);
                fi
        od;
        return alpha;
```

## 2.4. Specific search methods

Many search methods described above have been further specialized for particular applications. Some of these specializations have become widely known, and we can consider them as independent search algorithms. We present a graph that shows relations between these algorithms at the end of this chapter, and continue here with description of search algorithms.

### 2.4.1. A* algorithm

The *A* algorithm* is a concretization of the beam search algorithm for the problem of finding a shortest path in a graph. It is a good example of application of heuristics, because

in this case the heuristics can be explained and analyzed mathematically precisely. This algorithm was developed in 1968 (Hart, Nilsson, Raphael [17]) and researchers have since then several times showed interest in its analysis and improvement.

The problem it solves is *finding a shortest path* between two nodes of a graph, where besides the lengths of arcs of the graph also estimates of the lower bound of the distance to the end node are given for every node of the graph. We shall use the following notations:

*s* –initial node of the path;
*g* –end node of the path;
*p(n,i)* – length of the arc from the node *n* to the node *i;*
*l(n)* – length of the shortest path from the initial node *s* to the node *n;*
*h(n)* – length of the shortest path from the node *n to* the end node *g;*
*H(n)* – estimate of the lower bound of *h(n),* i.e. it should be $H(n) \leq h(n)$;
*open* – set of nodes to be checked;
*succ(n)* – successor nodes of *n*.

Like in beam search, this algorithm keeps an *open* set of nodes, and at each step finds in a loop the best continuation of a path. Then the lengths of paths to its successors are calculated and the set *open* is extended. This process continues until the end node *g* is reached.

**A.2.25:**

```
l(s) = 0;
open = {s};
while not empty(open) do
        x = selectFrom (open);
        n=x;
        for i∈ open\ {x} do
                if l(i) + H(i) < l(n) + H(n) then
                        n = i
                fi
        od;
        if  n == g then
                success()
        else
                for i∈ succ (n) do
                        l(i) = l(n) + p(n,i)
                od;
                open = open∪succ (n) \ {n};
        fi
od;
failure()
```

This algorithm has some interesting properties.

1. Under the conjecture that $H(n) \leq h(n)$ for each node *n* of the graph, this algorithm gives the precise answer – it finds an actual shortest path from *s* to *g*. Indeed, the inequality $H(n) \leq h(n)$ guarantees that, taking into account the condition $l(i) + H(i) \leq l(n) + H(n)$ in the algorithm, only those arcs will be ignored that can not be on the shortest path to *g*.

2. This is the fastest algorithm that can be constructed for solving the given problem precisely.

## 2.4.2. Unification

Unification is a method that is often used in AI programs. This is a search for expressions that will unify (make textually identical) given expressions when substituted for variables in these expressions. We shall need some definitions in order to speak about unification.

A *substitution* is a tuple of variable-expression pairs, for example:

*((x,A),(y,F(B)),(z,w)).*

We shall additionally require that no variable of a substitution can appear in an expression of the same substitution. A substitution describes a transformation of an expression where variables of the expression will be replaced by their substitute expressions. Application of a substitution *s* to an expression *E* is denoted by *E°s*. Example:

*P(x,x,y,v)°((x,A),(y,F(B)),(z,w)) = P(A,A,F(B),v).*

Substitutions *s* and *t* are independent, if their replaceable variables are different. In this case we can build a composition *s*t* of the substitutions *s* and *t* that is a substitution consisting of the pairs from both *s* and *t*.

If *E,...,F* are expressions, *s* is a substitution, and *E°s = ... = F°s* then *s* is a *unifier* of *E,..., F*. The *most general unifier* of *E,...,F* denoted by *mgu(E,...,F)* is the unifier *s* of *E,...,F* with the property that for any unifier *t* of *E,...,F* there exists a unifier *r,* such that *E°s*r=E°t, ... , F°s*r=F°t*. This unifier *s* is in a sense the least unifier of *E,...,F.* If there exists a *mgu(E,...,F)* for expressions *E,...,F,* then it is unique. This follows immediately from the definition of the most general unifier.

The expressions for unification can be terms, atomic formulas etc. Unification, i.e. finding of a *mgu,* can be directly used for answering queries to knowledge bases. For example, if we have a knowledge base about people that contains the facts:

*married(Peter, Ann)*
*married (Jan, Linda),*

then one can ask a question about Linda's marriage by asking to unify the formula *married(x,Linda)* with the facts of the knowledge base. The substitution *((x,Jan))* will show that *Linda* is married with *Jan.* Actually, we have used the unification already in resolution method and in Prolog in Chapter 1.

Before describing the algorithm of unification here, we shall agree on the syntax of expressions we use: a variable is an expression, and *(expression,...,expression)* is an expression. The first expression in parentheses is a function name, the other are arguments.

We are going to use the following notations:

*var(x)* – true, if *x* is a variable;
*const(x)* – true, if *x* is a constant;

*mguVar(x,y)* – gives *mgu* of the variable *x* and expression *y* (not a variable), and fails if *x* occurs in *y;*
*includes(x,y)* – true, if *y* includes *x*
*part(x,i)* – the *i*-th part of the expression *x*, i.e. the *i*-th element of the list that represents the expression *x*;
*subst(e,s)* -- applies substitution *s* to *e*;
*compose(g,s)* -- produces a composition of substitutions *g* and *s*.

A.2.26 is the *unification algorithm* expressed in the form of two functions *mgu* and *mguVar*. The function *mgu(x,y)* producesthe the most general unifier of two expressions *x* and *y,* if such exists, and returns empty list, if the expressions are not unifiable. The function *mguVar(x,y)* is a small function for unifying a variable *x* with an expression *y*.

**A.2.26:**

```
mgu(x,y)=
   if x==y then succes( )
        elif var(x) then succes(mguVar(x,y))
         elif var(y) then succes(mguVar(y,x))
        elif (const(x)∨const(y)) & x≠y ∨length(x)≠ length(y) then failure()
         else g = ( )
   fi;
   for i = to length(x) do
              s = mgu (part(x,i), part(y,i));
              if empty (s) then failure() fi;
              g = compose (g,s);
              x = subst(x,g);
              y = subst(y,g)
   od;
   success(g)
   fi

mguVar(x,y) =
        if includes(x,y) then
                failure()
        else
                return ((x)/(y))
        fi
```

## 2.4.3. Dictionary search

The problem considered here is to find a given word from a dictionary where the words are ordered in the lexicographic order. Additional knowledge is given about the frequencies of occurrences of letters in words. Let us denote the frequency of occurrence of the *i*-th letter of the alphabet by $p_i$. We do not consider dependency of these frequencies on the place of a letter in a word or on the neighboring letters, although it would be possible, and the algorithm would be in essence the same.

Total frequency of occurrence of letters preceding the *i*-th letter of the alphabet is

$$q_i = \sum_{j=1}^{i-1} p_j$$

If a dictionary contains $l$ pages, then an approximate value of the page number $h(w)$, where the word $w$ should be, can be computed knowing that $w$ consists of the letters with alphabetic numbers $i1,i2,...,in$, and the frequencies $p_j$ and $q_j$, $j=1,..,n$ of these letters are konwn:

$$h(w) = 1+l(q_{i1}+p_{i1}(q_{i2} + ... + p_{in}\, q_{in}) ...).$$

Let us introduce a function $f(x)$ of the page numbers of a dictionary which depends on the actual placement of words in the dictionary. This function will be defined as follows: a word $w$ is selected on a given page $x$. Its expected page number $h(w)$ is computed according to the formula given above. This page number is the value of $f(x)$.

Let us have the problem to find the page where the word $w$ is placed. We shall try to solve another problem which is close to it: to find the page $k$ where the words give the values of the function $h$ equal to $h(w)$. Now we have to solve the equation

$f(k)=c$, where $c$ is a constant, and $c=h(w)$.

Luckily, the function $f$ is monotonous, and we can use a simple method for solving this equation. We can use the *method of chords* for solving the equation $f(k)=c$ iteratively. The method computes a new value for one of the endpoints of the interval $[a,b]$ at every step of iteration, using a linear approximation of the function $f$ on the interval. This gives us the *dictionary search algorithm* A.2.27, see also Lavrov and Goncharova [23]. The algorithm has the following parameters:

$j$ – expected page number of the word $w$;
$a$ – number of the first page for search;
$b$ – number of the last page for search;
c – constant equal to $h(w)$.

The algorithm A.2.27 computes recursively a better approximation to $j$ that should be initially given equal to $c$, until the right value is found. In this case, *good(j)* means that the given word is on the page $j$. The beginning of algorithm is for the case when the value of $c$ is on the boundary of the interval of search (that is $j=a$ or $j=b$, and this happens at the final steps of the search).

**A.2.27:**

```
dict (c, j, a, b)=
        if good(j) then
                return j
        elif j==a then
                return dict (c, j+1,a+1, b)
        elif j==b then
                return dict (c,j-1,a, b-1)
        else
                 s= f(j);
                if s<c then
                        return dict (c,j + (c-s)*(b - j)/(f(b) - s), j, b)
                else
```

```
            return dict (c,a + (c-f(a))*(j - a)/(s – f(a)), a, j)
        fi
    fi
```

The main search is done in the ***else*** part of the algorithm. This part changes the expected result $j$ by $\Delta$ that is calculated depending on the condition $f(j) < c$. One step of iteration is shown in Fig. 2.5 for the case where the function $f$ is such that $f(j) < c$. In this case $\Delta$ is calculated as follows:

$$\Delta=(c-s)*(b - j)/(f(b) - s)$$

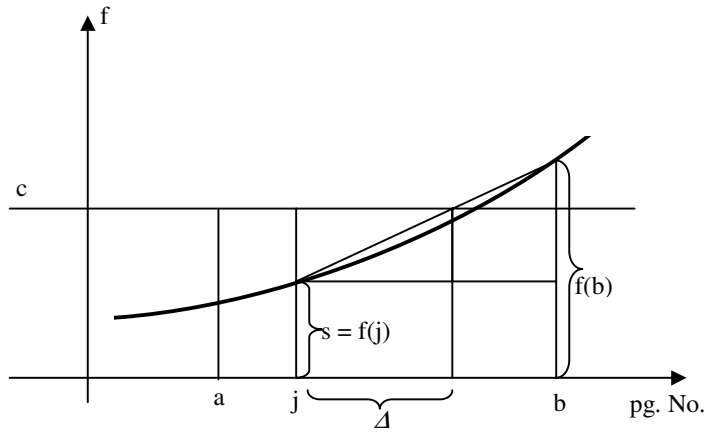



**Figure 2.5.** Method of chords

The algorithm can be easily improved, if one knows more about the probabilities of occurrence of letters in words. In particular, the first possibility will be to take into account the fact that probabilities of occurrence of first letters in words are different from probabilities of occurrence of letters in other positions. This will change only the calculation of $h(w)$ that will give a better approximation of position of a word in the dictionary.

This algorithm was developed when memories were small and binary search in a sorted file required many memory page changes. This algorithm can be useful today for search in very large dictionaries. It will rapidly decrease the part of dictionary where search must be done.

## 2.4.4. Simulated annealing

When a global optimum is looked for in a large search space with a multimodal fitness function, then local search methods do not give a good result. The reason is that they can only find a local optimum, and get stuck at it, because any step made from a local optimum to its neighborhood only spoils the value of a fitness function. To find another local optimum, and finally also a global optimum that is the best among the local optimae, one has to be able to make long steps and somehow use the information obtained during the search about the fitness function.

*Simulated annealing* is a search method that combines local search with stochastic global search for finding a probabilistically good approximation of global minimum. Its name and its basic idea originate from metallurgy where gradual cooling of a workpiece provides material with minimal internal energy. The cooling process must be done according to a given *schedule* prescribing how the temperature must be gradually reduced when the cooling occurs. The temperature determines how probable are random changes that do not give any local reduction of the fitness function, i.e. of the value (energy in metallurgy) that has to be minimized. It is known that a sufficiently slow cooling (long cooling time) gives global minimum with the probability as high as required (as close to *1* as required). The following notations are used in the algorithm of simulated annealing:

*s* – a reached sate;
*e* – energy of the state *s;*
*sb* – best state found;
*eb* – energy of *sb;*
*sn* – a neighboring state under consideration;
*en* – energy of *sn;*
*kMax* – maximal acceptable number of steps;
*sInitial()* – produces an initial state;
*neighborState(s)* – finds a neighbor sate of *s;*
*energy(s)* – calculates energy of *s;*
*random()* – generates random numbers evenly distributed between *0* and *1;*
*P(e,en,t)* – a probability of taking a random move in a global search space depending on energy *e* of current state, energy *en* of neighbor state and temperature *t;*
*temperature(k)* – temperature at the given step *k* of the schedule.

**A.2.28:**

```
s=sInitial;
sb=s;
eb=e;
k=0;
while k≤kMax do
        if good(s)then
                return s
        fi;
        sn=neighborState(s);
        en=energy(sn);
        if en<eb then
                sb=sn;
                eb=en
        else if random()<P(energy(s),en,temperature(k)) then
                s=sn
        fi;
        k=k+1
od;
failure()
```

The search operator *neighborState(s)* should produce statistically good local improvements of the state *s*, but must sometimes also make steps that not necessarily give

an improvement. In the latter case a step will be taken with a probability that depends on the schedule. This operator can be split in two separate operators: *localStep(s)* for local search, e.g. best first search, and *globalStep(s)* for random global search. Now the state *sn* is not necessarily a neighbor state to *s*. This gives another simulated annealing algorithm:

**A.2.29:**

```
s=sInitial;
sb=s;
eb=e;
k=0;
while k<kMax do
        if good(s)then
                return s
        fi;
        sn= localStep(s);
        en=energy(sn);
        if en<eb then
                sb=sn;
                eb=en;
                k=k+1;
        fi;
        sn= globalStep(s);
        en= energy(sn);
        if random()<P(energy(s),en,temperature(k)) then
                s=sn;
                k=k+1
    fi;

od;
failure()
```

It is known from the analogy with physics that the best schedule of changing the probability of global steps (i.e. the steps that do not improve the fitness value) should be exponentially dependent on the temperature $t$ and difference of fitness values $\Delta e$ of the current and the next step. The following formula that is used for calculating the probability $P$ of maikng a global step guarantees that this probability decreases as the temperature $t$ decreases (notice that $\Delta e<0$):

$$P(\Delta e,t)=\exp(\Delta e/t).$$

The temperature in its turn decreases with constant ratio less than 1 at each step and approaches $0$ in the limit, so does the probability $P$.

Looking at the physical explanation of then simulated annealing method, one can guess that this method can be implemented also on a parallel computing platform. Indeed, parallel implementations of simulated annealing exist and are efficient for specific applications like circuit design and combinatorial optimization.

## 2.4.5. Discrete dynamic programming

Discrete dynamic programming is a search method for stepwise building an optimal solution as a sequence of local solutions without backtracking. This is possible, if the search problem has the following properties:

- There is a finite number of possible choices at each search step.
- There is a finite number of partial solutions that can be considered as possible prefixes (initial parts) of the optimal solution at each step.

Originally, *dynamic programming* appeared as a method of selection of an optimal trajectory in a multidimensional space, e.g. selecting a path and fuel consumption program for bringing a spacecraft to orbit. This is obviously not a problem of finite search, but it can be reduced to finite search problem by substituting discrete values of coordinates instead of continuous values, as it is often done in numeric methods. This gave rise to *discrete dynamic programming*. The name dynamic programming was coined, and this method was thoroughly described, by Richard Bellman [3].

In the continuous case, one has the following condition instead of the second property of the discrete dynamic programming problem: given an optimal trajectory $T$ from $a$ to $b$ and a point $c$ on $T$, the part of $T$ that leads form $a$ to $c$ is the optimal trajectory from $a$ to $c$. This condition is fulfilled for *additive fitness functions* like a cost function, for instance, because value of an additive fitness function for a path is always a sum of its values for pieces of the path.

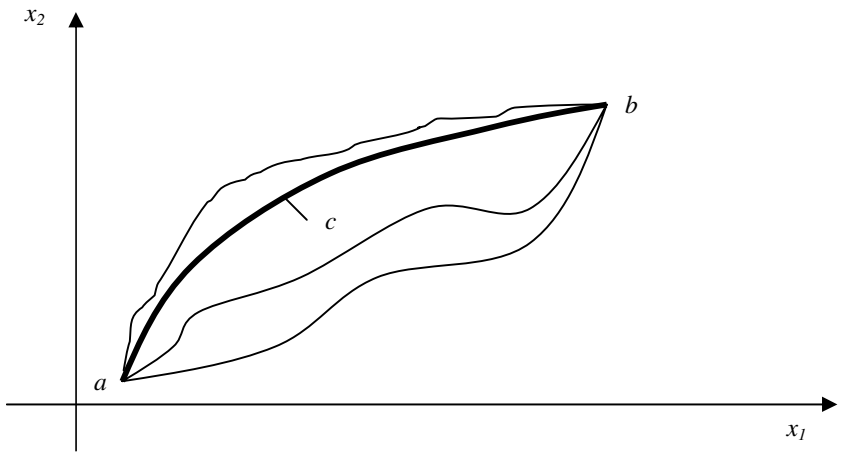



**Figure 2.6.** Dynamic programming – finding the best path

Fig. 2.6 illustrates a dynamic programming problem in a two-dimensional space with coordinates $x_1$ and $x_2$. One must find the best way to proceed from a given initial state $a$ to a given final state $b$. There is infinite number of ways to proceed. (Some possible ways are shown in figure.) A fitness function is given that assigns a numeric value for each trajectory. One must find the trajectory with maximal (or minimal) value of the fitness function. If the thick line is an optimal solution $T$ for the points $a$ and $b$, then the optimal

path from $a$ to any point $c$ on $T$ is the part of $T$ between $a$ and $c$, and not any other trajectory from $a$ to $c$. This is obviously not a problem of finite search, but it can be reduced to a finite search problem by using discrete values of coordinates instead of continuous values. This gives us the discrete dynamic programming problem.

Fig. 2.7 shows a *discrete dynamic programming step*. One knows the optimal paths from the initial state $a$ to intermediate states $s_1, \ldots, s_n$ at each search step. The aim is to find one step longer optimal paths from $a$ to the states $t_1, \ldots, t_m$ that follow the states $s_1, \ldots, s_n$. This can be done for each $t_i$, $i= 1,\ldots,m$ by trying out all possible continuations of the given partial optimal paths of $s_1, \ldots, s_n$ and finding the best among them.

In order to describe the algorithm of discrete dynamic programming, we use the following notations:

$path_s$ –optimal path from $a$ to $s$ constructed step by step; when $s=b$ is reached, the solution of the problem $path_b$ is found;

$k$ – number of a step on the path from $a$ to $b$, ($k=1,\ldots, maxStep$);

*reachedStates* – states reached by optimal partial paths on the step $k$, i.e. the set $\{ s_1, \ldots, s_n \}$ shown in Fig. 2.7;

*newStates* – set of states for extending the partial paths of the current step, i.e. it is the set $\{ t_1, \ldots, t_m \}$ shown in the figure;

*selectStates(k)* – creates the set *newStates* for the step $k$;

*f(x,y)* – value of additive fitness function of a path from state $x$ to state $y$; this function must be computable for all pairs of states of neighboring steps, and for a longer path it is the sum of its values for pairs of neighboring states along the path;

$h_s$ – value of fitness function for a partial path from $a$ to $s$. Let us notice that for the first step *reachedStates={a}* and for the last step *newStates={b}*.

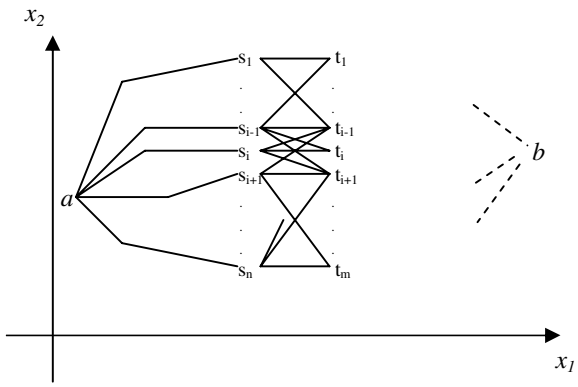



**Figure 2.7.** Discrete dynamic programming

The algorithm of discrete dynamic programming is the following:

**A.2.30:**

$path_a=();$

*reachedStates={a};*

```
h_a=0;
for k=1 to maxStep do
    newStates=selectStates(k);
    for t∈ newStates do
        sm=selectFrom(reachedStates);
        h_t= h_sm+f(sm,t);
        for s∈ reachedStates do
            h_temp= h_s+f(s,t);
            if h_temp > h_t then
                sm=s;
                h_t = h_temp;
            fi
        od;
        path_t=(path_sm ,t)
    od;
    reachedStates= newStates
od;
return path_b
```

Time complexity of this algorithm is $O(mr^2)$, where $m$ is the number of steps and $r$ is the number of possible states at each step. Indeed, at each step one has to check all pairs of neighboring states, i.e. $r(r-1)$ pairs.

## 2.4.6. Viterby algorithms

These algorithms were developed quite long ago and didn't get much attention first [49]. However, when data mining, and especially analysis of sequences in bioinformatics, became important, these algorithms gained popularity. This is a typical example of how algorithms of artificial intelligence are being introduced into practice.

A classical *Viteby algorithm* is an algorithm of solving a problem of so-called compound decision theory. It uses the *dynamic programming paradigm*, see Section 2.6. It is a word recognition algorithm, specifically for finding the most likely word for a sequence of patterns $x_1, \ldots, x_m$ that represent symbols (letters of the word). The goal is to choose the assignment $w_1, \ldots, w_m$, of letters from a given alphabet $L$ which has the maximum probability over all possible $r^m$ assignments, where $r$ is the number of letters of the alphabet $L$. The brute force search through all possible assignments has the time complexity $O(r^m)$. Assuming the *first-order Markov property* of the sequence of letters, i.e. that the probability of appearance of a letter in the sequence depends only on the preceding letter, we get an algorithm with time complexity $O(mr^2)$ instead of $O(r^m)$. Let us have the following notations:

$p1(x,w)$ – probability of the letter $w$ with the pattern $x$ being the first letter in a word;
$p(x,w,u)$ – probability of the letter $w$ with the pattern $x$ following the letter $u$ in a word;
$p_w$ – probability of the prefix ending with the letter $w$;
$pref_w$ – the best prefix ending with the letter $w$;
*selectFrom(L)* - produces an element of L.

The *Viterby algorithm* is as follows:

**A.2.31:**

```
for w∈ L do
    pW = p1(x1,w);
    prefW = (w)
od;
for i=2 to m do
        for wi∈ L do
            w= selectFrom(L);
            p = pW*p(xi ,wi,w);
            for z∈ L do
                    if pz*p(xi ,wi,z) > p then
                            w= z;
                            p = pz*p(xi ,wi,z)
                    fi
            od;
            pwi=p;
            prefwi=(prefW,wi);
        od
od
```

We should see the similarity of this algorithm to the algorithm of discrete dynamic programming from the previous section. The first loop in this algorithm is for handling the first patterns of the given sequence, i.e. making the first step of dynamic programming and initializing the prefixes of the optimal paths. The second loop is the main loop of the algorithm. It is similar to the discrete dynamic programming main loop. This gives the time complexity of the Viterby algorithm equal to the time complexity of discrete dynamic programming that is $O(mr^2)$.

An extended version of the algorithm is *Dictionary Viterbi Algorithm* (DVA) that takes into account also a dictionary – only words from the dictionary are accepted. We use the following additional notations here:

*inDictionary(z)* – is true, iff there is a word beginning with the prefix $z$ in the dictionary;

*selectFrom(pref,L)* gives a letter that suits for the prefix *pref,* so that the prefix is in dictionary.

**A.2.32:**

```
for w∈ L do
        pW = p1(x1,w);
        prefW = (w)
od;
for i=2 to m do
    for wi∈ L do
        w= selectFrom(prefwi,L);
        p = pW*p(xi ,wi,w);
        for z∈ L do
                if pz*p(xi ,wi,z) > p and inDictionary(prefW,wi) then
```

$w = z;$
$p = p_z * p(x_i, w_i, z)$
**fi**
**od;**
$p_{w_i} = p;$
$pref_{w_i} = (pref_w, w_i);$
**od**
**od**

There is a problem with the function *selectFrom(pref,L).* It may happen that no letter will be found from *L* that together with the *pref* gives a prefix from dictionary. Hence, the algorithm A.2.32 does not always give the expected result, unless the function *selectFrom(pref,L)* selects a really good continuation every time. In order to avoid this situation one should introduce backtracking into the algorithm. The backtracking step must be done at the point where the function-call *selectFrom*($pref_{w_i}$,*L)* shows that no prefix is available. This requires to introduce a recursive call and additional parameters of the whole algorithm, see Section 2.3.5. However, the situation with the backtracking in this case is worse than it may seem at the first glance. Backtracking spoils the required additive property of the fitness function calculated along a partial path, so that dynamic programming cannot be applied any more.

## 2.5. Forward search and backward search

Quite often, a search problem is finding a path from an initial state to a final state. Both these states are given, and solution of the search problem is a path from the initial state to the final state. Problems of this kind appear in proof-search, planning and program synthesis. The solution is built usually gradually, extending step by step a partial solution like it has been described in Section 2.3.5 for search with backtracking. This problem can be solved in two ways:

- by *forward search*, starting the search from the initial state and building a path stepwise in the forward direction towards the final state;
- by *backward search*, starting the search from the final state and building a path in the backward direction towards the initial state.

A question, which search direction is better, can be often answered by the analysis of branching factor of a search tree of the problem. Obviously, as the number of steps taken along the path is the same in both cases, it is reasonable to select the search direction that gives a search tree with smaller branching factor and smaller backtracking. Let us consider as an example the calculus of computability described in Section 1.1. If the axioms of a calculus of computability have only one output and several inputs, e.g. are of the form

$\{a,\ldots,b\} \rightarrow \{c\},$

then it is obviously reasonable to perform forward search, because, in this case, no backtracking is needed, and branching factor in forward direction is less than in backward direction. To the contrary, if the axioms have only one input and several outputs, and outputs of axioms do not intersect, then it is reasonable to perform search in backward

direction, because then the branching factor will be less, and no backtracking will be needed. If the outputs of axioms have common elements in the second case, then one has to take into the account backtracking as well, and selecting the best search direction is more difficult.

Sometimes it is reasonable to start search from both sides and try to meet somewhere in the middle. If the numbers of steps made in both directions are more or less equal, and branching factors are also equal in both directions, then the total number of search steps may be considerably less than in a unidirectional search.

## 2.6. Hierarchy of search methods

The search methods considered here constitute a hierarchy shown in Fig.2.8. On the upper level of the hierarchy these methods can be divided into exhaustive search and heuristic search methods. Simple exhaustive search is represented by breadth-first and depth-first search methods. But there are other more complex methods of exhaustive search, including dynamic programming and search on and-or trees (and-or-search). Backtracking can be applied both for exhaustive search and for heuristic search.

A very clear inheritance path goes from search in general to exhaustive search, dynamic programming, Viterby search and dictionary Viterby search. Depth-first search has many concretizations, and we have touched only some of them.

The most widely used search methods are probably unification and binary search. The latter can be used both in exhaustive search and in heuristic search. The ulr and url depth-first search methods are binary search methods, although we have not shown this in the figure.

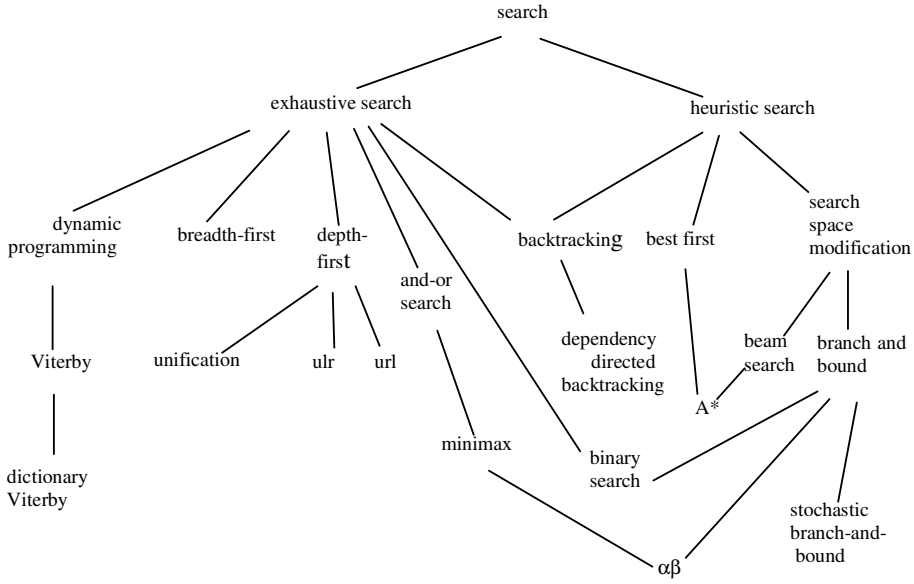



**Figure 2.8.** Hierarchy of search method

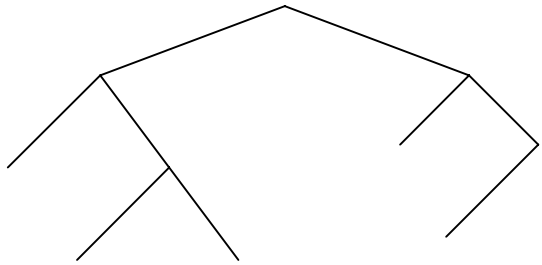
Heuristic search methods that have been considered here include backtracking, search with search space modifications and best-first search. These basic heuristic search methods give rise to many widely used specific heuristic search algorithms. An example is the A* algorithm that is a specialization of beam search and best first search. An important example is alpha-beta pruning that is a specialization of minimax and branch and bound search, and is widely used in chess programs. We have only very briefly considered a class of stochastic search methods, discussing only the stochastic branch-and-bound search.

The search methods discussed here will be extensively used in the algorithms of learning, planning and problem solving considered in the following chapters.

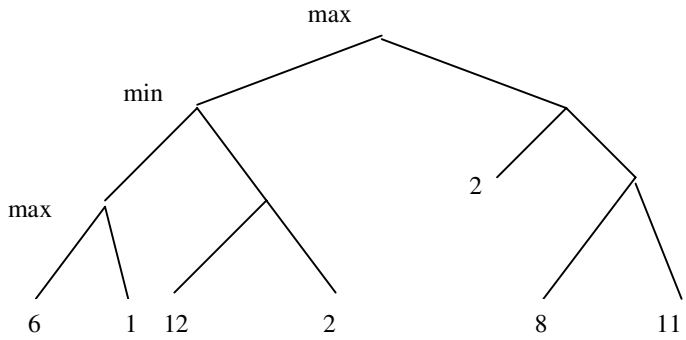
## 2.7. Exercises

1. Define a search problem of finding a product in an unfamiliar foodstore. Specify a search space. Which heuristics do you use in this case?

2. Label the nodes of the following binary tree with natural numbers for the search order *lur* (i.e. in the orer left-upper-right):



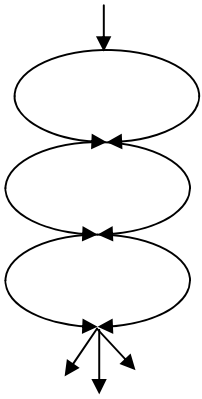
3. Use alpha-beta pruning on the game tree given below with gains written at leaves. The first move is done by a player that tries to get a maximal gain. Its opponent tries to end the game with minimal gain. Write at nodes the expected gain (alpha or beta). Show the nodes that will not be checked by alpha-beta search.




4. How many states must be remembered for breadth-first search to the depth 5, if the branching factor of a search tree is 4?

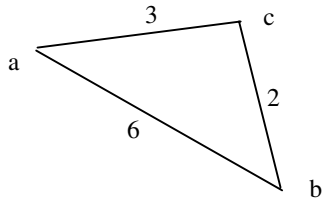
5. How many sums (of path lengths) must be computed in discrete dynamic programming, if there are 10 possible states at each step and the number of steps is 10?

6. Draw a search tree for the following search graph. (The search graph includes all possible states and paths accessed during the search.

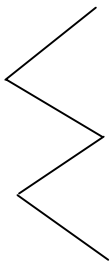


7. What is the time complexity of discrete dynamic programming algorithm with repect to the number of steps and maximal number of states at a step?

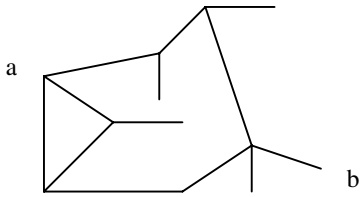
8. Explain, which actions will be taken by A* algorithm to find the shortest path from the node *a* to the node *b* in the following graph, where the distances between its nodes are shown on its edges:




9. Write binary search programs for the search orders *lur, rul, lru* and *rlu (see Section 3.2.3)*. Show the order of traversing the following binary tree in each of these cases:



10. Draw a search tree for traversing the labyrinth from point *a* to point *b*.




11. Draw a search graph for the same labyrinth-traversing problem. Compare the search tree and the search graph.

This page intentionally left blank

# 3. Learning and Decision Making

Learning is improving a knowledge system by extending or rearranging its knowledge base or by improving the inference engine. This is one of the most interesting problems of artificial intelligence that is still under development. Machine learning comprises computational methods for acquiring new knowledge, new skills and new ways to organize existing knowledge ([46], [33]).

One can distinguish three stages of development of machine learning. In the early days of AI research, simple systems of parametric learning were developed and used for building adaptive systems. The role of predefined knowledge and symbolic learning was understood later. New approaches to massively parallel learning appeared when high-performance computing started and massively parallel hardware was developed.

Donald Michie has proposed the following criteria of learning which enable one to distinguish several different levels of learning [34]:

- *weak learning* is improving problem solving abilities by using examples;
- *strong learning* criterion requires additionally an ability to present the new (learnt) knowledge explicitly in the form which is understandable for a third party, i.e. for other actors besides the learner and tutor;
- *very strong* learning comprises the requirements of the strong learning together with the requirement that the acquired knowledge has to be operational for a third party.

Problems of learning vary greatly by their complexity from simple parametric learning which means learning values of parameters, to complicated forms of symbolic learning, for example, learning of concepts, grammars and functions. A distinguished class of learning methods present parallel learning algorithms that are suitable for execution on parallel hardware. These learning methods are represented by genetic algorithms and neural nets.

## 3.1. Learning for adaptation

### 3.1.1. Parametric learning

Parametric learning algorithms were developed already in the early days of computing. They mimic simple adaptive behavior in living creatures. Let us consider the case where choice must be made between two cases that we denote by 0 and 1. Let the probability of making the choice 1 be $p$, and for choosing 0 it will be $q$. The learning algorithm will change these probabilities, depending on the encouragement or discouragement that the algorithm receives, preserving the constraint

$$p+q=1$$

for the probabilities. It is reasonable to choose a law for changing $p$ and $q$ so that in the case of permanent encouragement (discouragement) the probability $p$ (respectively $q$) will asymptotically approach 1. The following formula is suitable for increasing the probability $x$ of a right decision:

$$x'=x+a(1-x),$$

where $x'$ is the new probability. The small positive constant $a$ determines the intensity of change of probabilities. Considering the equality $p+q=1$, we can calculate the new

probabilities *p'* and *q'* in the case of encouragement of the choice with the probability *p* from the formulas

*p'=p+a*q*
*q'=1-p*

and in the case of discouragement from the formulas

*q'=q+a*p*
*p'=1-q.*

This gives us the algorithm A.3.1 for *parametric learning*, implemented here as a very simple Java class *Learner*. This class has methods *encouragement* and *discouragement* that modify the probabilities *p* and *q* as stated above. Its constructor gets an initial value of the probability *p* and a value of the constant *a* that determines the learning speed.

**A.3.1:**

```
class Learner{
        int decision;
        double p, q, a;
        Learner(double p, double a){
                this.a=a;
                this.p=p;
                q=1-p;
        }
        void encouragement(int decision) {
                if (decision==1) {
                        p=p+a*q;
                        q=1-p;}
                else {
                        q=q+a*p;
                        p=1-q;
        }
        void discouragement(int decision) {
                if (decision==1){
                        q=q+a*p;
                         p=1-q ;}
                else {
                        p=p+a*q;
                        q=1-p;
                }
        }
}
```

## 3.1.2. Adaptive automata

A behavior similar to *learner* can be expressed by a finite automaton that has a simple transition graph with linear branches for external states. This automaton is called *linear automaton*, because of the structure of its transition graph, shown in Fig. 3.1. Its output

depends on the branch of the transition graph where its current state is. When it gets encouragement as an input, it moves further away from the center, and vice versa, if it gets discouragement, it moves closer to the center. It changes the branch, if the center has been reached. Its memory, i.e. how strongly it can remember the branch where it is, depends on the number of states in one branch. From the other side, longer branches make an automaton more conservative – it will adapt slower to changes of the environment, because it takes more steps to change a branch when it gets discouragement.

This automaton is implemented by the Java class *LinAutom* that is shown as the algorithm A.3.2. This class has methods *encouragement* and *discouragement* like the *Learner* class. Besides that, it has a method *getState()* for determining its state. The constructor of this class gets number of states *numOfStates* that determines the length of a branch, and it assigns initial value 1 to the *state*.

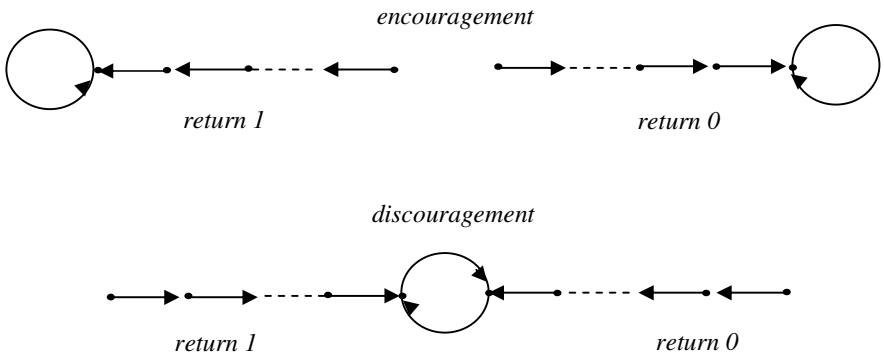



**Figure. 3.1**. Transition graph of a linear automaton

**A.3.2:**

```
class LinAutom{
        int state,maxState;
        LinAutom(int numOfStates) {
                maxState=numOfStates;
                state=1;
        }
        void encouragement(){
                if (abs(state) < maxState) {
                        state=state+sign(state);
                }
        }

        void discouragement() {
                 if (abs(state) > 1) {
                        state=statestate - sign(state);}
                else {
                        state=state-2*sign(state);
                }
```

```
        }
        int getState() {
                return state;
        }
}
```

The constant *maxState* determines here the stability of decisions of the automaton, i.e. it plays a role analogous to constant *a* in the *learner*. The values of *a* and *maxState* must be adjusted to the environment where the learning occurs - the more stable the environment the smaller can be *a* and the larger can be *maxState*.

This automaton can be generalized for choosing between *n* different output values, i.e. between *n* different decisions. Fig. 3.2 shows the transition diagram of such an automaton for *3* decisions. In the case of encouragement this automaton behaves very much in the same way as the linear automaton with two outputs. However, to find the right decision in the case of discouragement (to find the right branch of the transition graph) may take several steps even if the automaton already is in one of the central states.

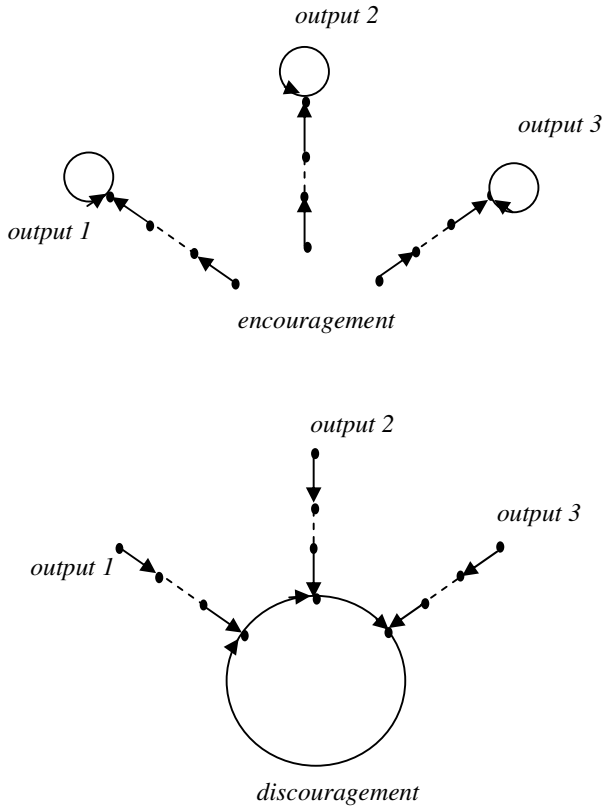



**Figure. 3.2.** Learning automaton for 3 alternatives

# 3.2. Symbolic learning

## 3.2.1. Concept learning as search in a hypothesis space

*Symbolic learning* of concepts is performed by means of constructing hypotheses and validating the *hypotheses* on the basis of available knowledge. In essence, it is a search in a *hypotheses space* that includes all possible hypotheses. The analogy between search and learning of concepts is quite obvious for concept learning from examples and counterexamples.

Let us have a set of properties that can be present or absent in individual objects. We wish to classify these objects on the basis of their properties and we shall say that a description of a class is a candidate concept, or a *hypothesis*, if it has not been proved to be correct yet. The goal is to build concepts from a given set of objects with known classes.

For a fixed set of properties, we can describe each object by a binary vector, denoting by 1 in the *i*-th position the presence and by 0 in the *i*-th position the absence of the *i*-th property. A *concept* can be described by one or more ternary vectors where 1 and 0 denote also presence or absence of a property, and besides that, a third value for which we choose *, will denote the fact that the particular property is inessential in the vector, i.e. it can be either 1 or 0.

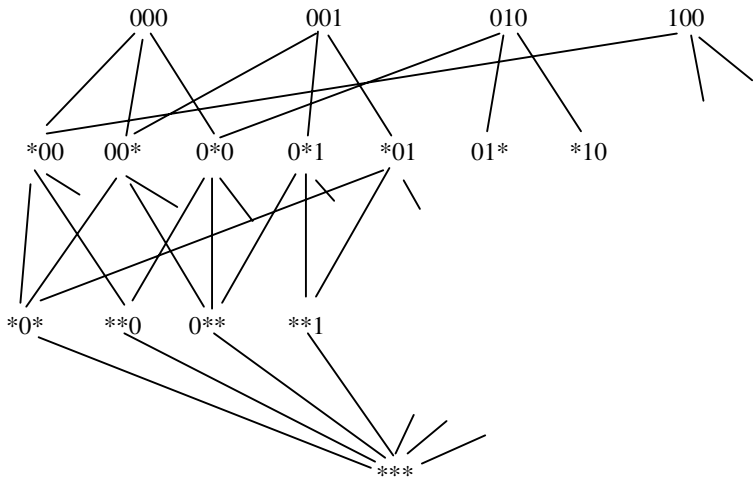



**Figure 3.3.** Hypotheses space

The search space for concepts is a set of ternary vectors, and it has several useful properties. The vector containing * in each position represents the most general concept which covers (includes) all objects. The vectors which do not contain * at all represent particular objects. Inclusion of concepts (and objects) is easy to detect by comparing the vectors. If a vector *v1* contains in every position which is different from * the same value as a vector *v2* then we say that *v1* covers *v2*. In this case, each object that belongs to the concept represented by *v2* (that matches *v2*) belongs also to the concept represented by *v1*.

Fig. 3.3 shows us a part of the search space for objects with three properties. Inclusion of concepts (hypotheses) is shown by lines connecting the vectors. Examples are the vectors without *. We see that the hypothesis 00* matches the examples 000 and 001. The same examples match the hypotheses 0**, *0* and ***. If we get a counterexample that matches a hypothesis, then we must strengthen the hypothesis (change it so that the counterexample will not match the hypothesis). The counterexample 010 matches the hypotheses 0** and *** and its occurrence shows that we must take the hypothesis 00*, but not 0** or ***.

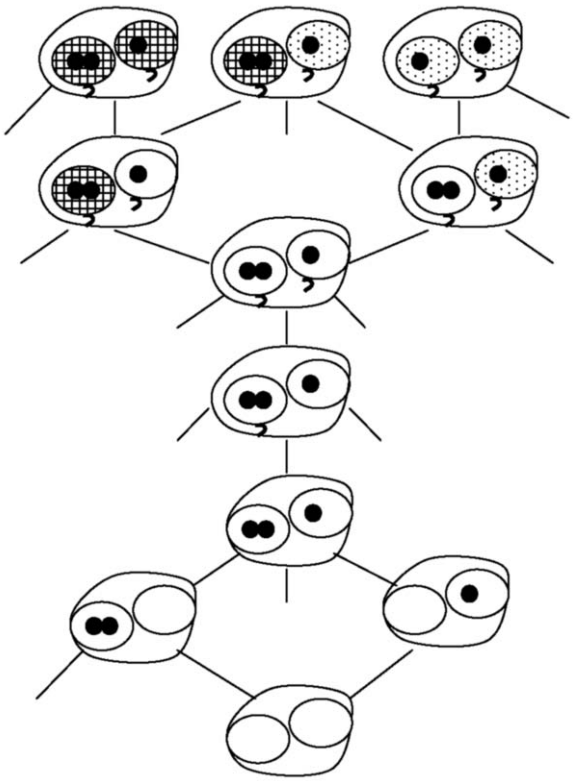


**Figure 3.4.** Part of a hypotheses space for classification of cells

Having a set of examples, it makes sense to select the least general hypothesis as the candidate for the concept. For our two examples this can be only 00*. But in many cases the choice is still not unique. This leads us to a search in the hypothesis space.

In general, one may need more than one vector for describing a concept. If the positive examples are 000, 001, 010, 100 and the negative examples (counterexamples) are 111,110,101,011, then we cannot describe the concept by means of one single vector. A possible description will be by the following three vectors: 00*, 0*0, *00. However, if we

could add a new attribute: "2 properties absent" at the fourth position then this concept could have been described by the single vector ***1.

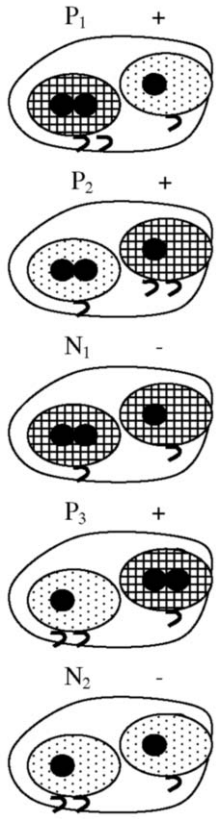



**Figure 3.5.** A set of examples for concept learning

To illustrate the symbolic learning algorithms we will use a more meaningful example, taken from [46]. In this example, the objects are imaginary cells that can be healthy or contaminated. The goal is to discriminate healthy and contaminated cells. Part of the hypothesis space is shown in Fig. 3.4. The cells have one or two nuclei, two different levels of darkness and can have one or two tails. The absence of coloring, nuclei or tails shows that the respective property is inessential in the particular case. The partial ordering of the hypotheses space is also well seen in Fig. 3.4. We shall discuss two different learning algorithms on this example: specific to general and general to specific concept learning.

## 3.2.2. Specific to general concept learning

The *specific to general concept learning* algorithm described here uses both *positive examples* (the objects that belong to the concept) and *negative examples* (the objects that do

not belong to the concept). It starts from any positive example and takes it as a hypothesis (i.e. a possible description of the concept). If a new positive example appears, then the hypothesis is generalized so that it covers the new example as well. If a negative example appears, it is added to the set of all negative examples that must be taken into account when hypotheses are being built – no hypothesis should cover a negative example.

Let $H$ be the current set of hypotheses, and let the set of negative examples observed be $N$. The learning algorithm proceeds as follows:

1. At the beginning, $H$ can be initialized to any given positive example and $N$ to the empty set.

2. If $p$ is the next positive example, then the algorithm proceeds in the following way:

- For each $h \in H$ that does not match $p$, replace $h$ with the most specific generalization(s) of $h$ that will match $p$.
- Remove from consideration all hypotheses that are more specific than some other hypothesis in $H$.
- Remove from $H$ all hypotheses that match a previously observed negative example $n \in N$, since they are overly general.

3. If $n$ is a new negative example, then the algorithm proceeds in the following way:

- Add $n$ to the set of negative examples $N$.
- Remove from $H$ all hypotheses that match $n$ since they are overly general.

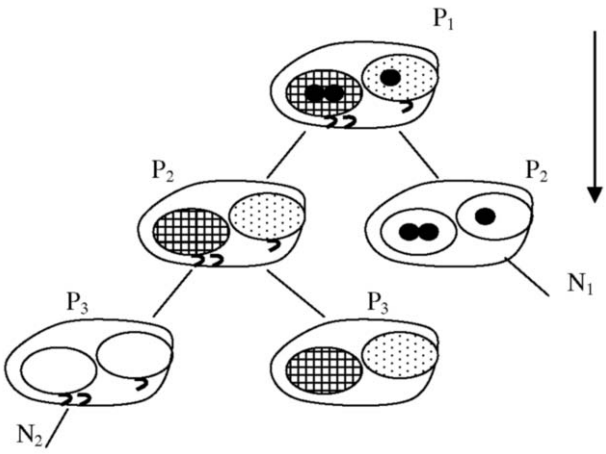



**Figure 3.6.** Example of specific to general concept learning

If the sequence of examples shown in Fig. 3.5 is presented to this algorithm, then the hypotheses shown in Fig. 3.6 will be generated in the order indicated by the arrow. The first positive example is taken as a hypothesis. The second example is positive as well. It allows

us to generalize the hypothesis in two ways. First, we can drop the number of nuclei in cells and keep color and tails. Second, we can drop nuclei and tails and keep coloring (dark or light). We get two branches of a search tree for the concept. The third example is negative and it covers the hypothesis without colors. This closes the search along the branch without colors. The next example is positive again, and we get two branches: one with tails and another with colors. The next negative example closes the branch with tails. As the result, the concept "one dark and one light" will be generated, as shown in Fig. 3.6.

In order to present the algorithm in more detail, we introduce the following notations:

*gen(p,h)* – set of the most specific generalizations of *h* which cover *p*;

*prune(ΔH,N)* – removes from *ΔH* all the hypotheses that cover some negative instance from *N*;

*match(p,h)* – a predicate which is true iff *h* covers *p;*

$y<x$ denotes that the hypothesis *x* covers the hypothesis *y*.

The algorithm is represented by two functions *addPositive* and *addNegative* shown in A.3.3 and A.3.4.

**A.3.3:**

```
addPositive(p,H,N)=
    if empty(H) then
        H={p}
    else for h ∈ H do
        if not match(p,h) then
            ΔH=gen(p,h);
            for x ∈ ΔH do
                for y∈ H do
                    if y<x then
                        H=H\{y}
                    fi
                od
            od
            prune(ΔH,N);
            H=H ∪ ΔH\{h}
        fi
    od;
fi
```

**A.3.4:**

```
addNegative(n,N)=
    N=N ∪{n};
    prune(H,{n})
```

## 3.2.3. General to specific concept learning

The algorithm that we describe here is a dual algorithm of the specific to general concept learning algorithm presented in the previous section. It starts from the most general concept, taking it as a hypothesis that covers all possible objects. When a negative example

appears, the algorithm builds the most general specialization of the hypothesis that will not cover the new negative example. Positive examples are collected into a set that is used for testing the hypotheses for sufficient generality.

Let *H* be the current set of hypotheses and *P* be the set of positive examples. The algorithm works as follows:

1. At the beginning, *H* can be initialized to include only the most general concept and *P* to the empty set.

2. When a negative example appears, then the algorithm proceeds in the following way:

- Each hypothesis that matches a negative example will be replaced by its most general specialization(s) that will not match the negative instance.
- All hypotheses that are more general than some other hypothesis will be removed.
- The hypotheses that fail to match all positive examples will be removed, since they are overly specific.

The first step can be accomplished by finding differences between the negative example and some positive examples associated with the hypothesis.

3. When a positive example appears, then the algorithm does the following:

- The positive example will be added to the set *P* of positive examples.
- All hypotheses that fail to match the new positive example will be removed from the set *H* of hypotheses.

We use the following notations for describing the algorithm:

*spec(n,h)* – set of the most general specializations of *h* that do not match *n*.
*prune(dH,P)* – removes from *dH* all hypotheses that fail to match some positive example in *P*.
*match(n,h)* – a predicate which is true if the hypothesis *h* covers the example *n*.
$y<x$ denotes that the hypothesis *x* covers the hypothesis *y*.

The algorithm of *general to specific concept learning* is presented by two functions a*ddNegative* and a*ddPositive* shown in A.3.5 and A.3.6. At the beginning of learning, the hypothesis set *H* must be initiated. It must initially contain the most general concept that covers all possible examples.

**A.3.5:**

```
addNegative(n,H,P)=
    for h ∈ H do
        if match(n,h) then
            dH=spec(n,h);
            for x ∈ dH do
                for y ∈ H do
                    if x < y then
```

```
                                        H=H \ {y}
                                    fi
                                od
                            od
                            prune(dH,P);
                            H=H ∪ dH\ {h}
                        fi
                od
```

**A.3.6:**

```
addPositive(p,H,P)=
    P=P ∪ {p};
    prune(H,{p})
```

The algorithms presented here for concept learning are applicable only in the case when a concept can be represented by a single hypothesis. In other words, the hypothesis space must include elements that represent the whole concept. This is a restriction that can be removed, and we will do it later.

## 3.2.4. Inductive inference

We present now the description of an algorithm that has a precise logical meaning, and is a basis of several other learning algorithms. Let us consider a problem where the goal is to learn a primitive recursive function from examples that are pairs of argument and function values. This is an *inductive inference problem*, where one has to generalize on the basis of partial information.

Primitive recursive functions have descriptions put together from very simple pieces. For instance, we can take the constant *0* (zero), operation *+1* (plus one) and an iterator *I* which prescribes a repetition of computations described as an argument of *I* the number of times given as its another argument. This set of building blocks is sufficient to build a description of any *primitive recursive function* as a superposition of these building blocks. As a consequence, the set of all primitive recursive functions is enumerable. This means that one can start building these functions in some regular way, starting from those that have the simplest (shortest) descriptions and continue the process as long as needed to reach the description of any particular primitive recursive function. Besides that, the primitive recursive functions are total functions – when applied to any natural number, they will give an answer. More about recursive functions can be found in textbooks, for instance [8].

**Theorem:** There exists an algorithm which, for any given sequence of pairs $(x_1,y_1)$, $(x_2,y_2)$, ... , $(x_n,y_n)$, constructs a primitive recursive function that computes values $y_1$, $y_2$, ... , $y_n$ for arguments $x_1$, $x_2$, ... , $x_n$ respectively.

The algorithm A.3.7 presented below becomes a proof of this theorem, if one takes *firstFun()=0* and *nextFun(f)* suitable for adding one by one new operations to $f$ in such a way that all possible functions will be built.

As we see, this theorem from logic ensures us that we can learn primitive recursive functions from examples. Inductive inference is even more general – one can learn from examples also other kinds of objects that belong to an enumerable set of constructive

objects. For instance, it is possible to learn grammars, theories, programs in some restricted languages etc. from examples.

The algorithm A.3.7 is a general form of an *algorithm of inductive inference* that finds the first function that computes values $y_1, y_2, \ldots, y_n$ for arguments $x_1, x_2, \ldots, x_n$ respectively. It uses the function *nextFun(f)* for getting the function next to the given function *f*, and the function *firstFun()* for taking the first function under consideration. This algorithm is actually the brute force search algorithm described in the beginning of Chapter 2. It is expected that the algorithm starts from the simplest candidate function given by *firstFun*, and generates the candidate functions with increasing complexity, that guarantees that exhaustive search is done.

**A.3.7:**

```
f=firstFun();
do
        L:{for i to n do
                if f(x_i) ≠ y_i then break L fi;
          od;
          success(f)
          }
          nextFun(f)
od
```

The whole complexity of computations is hidden in the function *nextFun* that must take into account also the information obtained from previous unsuccessful attempts as well as other useful knowledge about the class of objects to be constructed. A good *nextFun* builds the function *f* gradually, using already constructed parts in a new candidate function, and doing backtracking if needed.

It is essential for successful learning that the set of examples should be representative. For instance, if we give the sequence of pairs (1,2), (2,3), (3,4), we will get the function $y=x+1$ as the result of any reasonable inductive inference algorithm, because $x+1$ is the simplest function that fits the examples. However, if we say that we expected some other function, let us say, such that covers also the pairs (1,2), (2,3), (3,4), (4,6), then the fault is ours, because the examples (1,2), (2,3), (3,4) are not representative in this case.

## 3.2.5. Learning with an oracle

The learning capabilities can be improved by using an oracle that will answer questions asked by a learning algorithm when additional information is needed. For instance, when learning a concept only from positive examples, an overgeneralization may happen, which is impossible to detect without additional information. In such a case, an oracle can be asked whether the instances satisfying the hypothesis are also correct instances of the class being learnt. If the answer will be negative, then the hypothesis must be changed in the same way as in the case of a negative example satisfying the hypothesis.

An algorithm that learns with the help of an oracle from positive examples and background knowledge (that can be given in advance as well as gradually obtained during learning) is described in the following terms:

*examples* – examples for learning;
*known* – background knowledge;
*generalize(e,k)* – generalizes examples *e* by using background knowledge *k*

and produces a new hypothesis;

*instantiate(h,e)* – uses facts from examples *e* and builds a new instance of the hypothesis *h;*

*askOracle(i)* – asks an oracle whether the instance *i* is correct.

A general algorithm of learning with an oracle is the following:

**A.3.8:**

```
do
        hyptohesis=generalize(examples,known);
        instance=instantiate(hypothesis,examples);
        accept=askOracle(instance);
        if accept then success() fi
od
```

The function *generalize()* can use the specific to general learning algorithm described in Section 3.2.2. More than one good instance may be generated before the oracle accepts a hypothesis..

## 3.2.6. Inductive logic programming

It is possible to learn by inductive inference new clauses, new predicates and new Prolog programs. This is called *inductive logic programming* (ILP). The clausal form provides a good structure of objects, and already known clauses can be used as *background knowledge* that can help to guide the search needed in inductive inference.

Let us look at a family relations example where we already have the following background knowledge:

```
parent(pam,bob)        female(pam)
parent(tom,bob)        female(liz)
parent(tom,liz)        female(ann)
parent(bob,ann)        female(pat)
parent(bob,pat)
parent(pat,jim)
```

Our goal will be to learn a concept of daughter, i.e. to define the predicate

```
hasDaughter(X)
```

Let us have the positive examples

```
hasDaughter(tom)
hasDaughter(bob)
```

and negative examples

```
hasDaughter(pam)
hasDaughter(jim).
```

We see that the constants *tom* and *bob* which occur in positive examples (they have a daughter) do not occur in the predicate *female*, but occur in the predicate *parent*, hence a definition of the predicate *hasDaughter* must include at least the predicate *parent*. So we can already build a part of the definition:

*hasDaughter(X):- parent(X,Y),…*

Now we see that a new variable *Y* has appeared in the definition. This variable must be somehow constrained in the definition. The simplest guess will be to check the background knowledge about *female*. Using positive and negative examples, we can indeed find that the clause

*hasDaughter(X) :- parent(X,Y), female(Y)*

covers all positive examples and does not cover any negative example. This is the correct answer. One can check it by generating more examples and asking an oracle whether the examples are correct.

A set of algorithms of inductive logic programming has been developed as descendants of the program FOIL developed in 1990 by Quinlan [41]. The FOIL system applies general to specific concept learning for deriving a collection of clauses that define a predicate. The predicate's signature (its name and parameters) is given in advance as a goal, for instance, in the example above a goal was *hasDaughter(X).* This predicate becomes a head of a Prolog clause – a positive literal of the clause. In order to present the FOIL algorithm, we have to introduce a number of notations, some of them express quite intricate functions that include search:

*ex* – a set of positive and negative examples;
*target* – a literal for the goal predicate;
*clauses* – a gradually built set of clauses;
*newLiterals(c)* – constructs all possibly useful new literals to add to the clause *c*;
*newClause(ex,target)* – generates a new clause that covers some examples from *ex* and has the positive literal *target*;
*chooseLiteral(ls,ex)* – selects a literal from the set *ls,* taking into account *ex*;
*extendExample(e,l)* – creates a set of examples by extending the example *e* with all possible constants for variables of the literal *l;*
*covers(c,e) – true,* if clause or literal *c* covers example *e;*
*includesNeg(ex) – true,* if the set *ex* includes some negative examples;
*includesPos(ex) -- true,* if the set *ex* includes some positive examples.

An important function is *chooseLiteral(ls,ex)* that decides which literal to select next from the set of possible literals *ls*. This function uses the set of examples *ex* to decide about the suitability of literals. The quality of the whole algorithm depends very much on this function. In our example above we had some clues to decide about the selection of the literal *parent(X,Y)* first and *female(Y)* thereafter.

The FOIL algorithm is presented by two functions:

- *newClause(ex,target)* constructs a new clause from a given set of positive and negative examples *ex* and a given goal called here *target;*
- *foil(ex, target)* constructs a set of clauses calling *newClause* while there are still uncovered positive examples.

The main work is done by the function *newClause*. It works as long as there are negative examples covered by the clause that is constructed gradually, adding new negative literals one by one to the clause. After adding a new literal the algorithm has to generate new examples by substituting constants instead of the variables of the new literal. Algorithm A.3.9 presents the FOIL system.

**A.3.9:**

```
foil(ex, target) =
        exex=ex;
        caluses={};
        while includesPos(exex) do
                clause=newClause(exex,target);
                for e∈ exex  do
                        if covers(clause,e)then exex=exex\{e}
                od;
                clauses= clauses∪clause
        od;
        return clauses


newClause(ex,target) =
        clause=(target);
        exex=ex;
        while  inludesNeg(exex) do
                l= chooseLiteral(newLiterals(clause),exex);
                clause=(clause,l);
                eX=exex;
                for e ∈ exex  do
                        if covers(l,e)then
                                eX=eX∪ extendExample(e,l)
                        fi
                od;
                exex=eX
        od
```

## 3.2.7. Learning by inverting resolution

Learning by inverting resolution is a method of inductive inference for *learning a theory from examples*. A theory that is learnt is presented in the form of Horn clauses. In the process of learning, background knowledge is used, and this knowledge is also represented in the form of Horn clauses.

In this section we need some familiarity with the resolution method of derivation of clauses presented in the first chapter. A clause is a collection of positive or negative literals that we denote by capital letters with or without the negation sign "-": *A, -A* etc. Let us remind that resolution is applicable for deriving a new clause *C* from given clauses *C1* and *C2* by resolving them on a literal *L1* from *C1* and a literal *L2* from *C2*, if there exists a substitution *s* such that $L1 \circ s = -L2 \circ s$ where $L1 \circ s$ and $L2 \circ s$ denote the result of applying the substitution *s* to *L1* and *L2*. We shall use only Horn clauses here. They contain at most one

positive literal, and can be represented in the form $A \leftarrow B,...,D$ where $A$ is the positive literal and $B,...,D$ are negative literals. We also make an assumption that, for any resolution step, the resolved clauses do not contain common literals except the literals resolved.

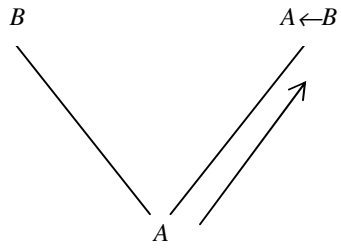



**Figure 3.7.** Inverting the resolution

Let us consider the resolution step shown in Fig. 3.7. If the literals are propositional variables, then we can reconstruct the clause $A \leftarrow B$ from the resolvent $A$ and the premise $B$, and we can also reconstruct the clause $B$ from $A$ and $A \leftarrow B$. This is using a resolution in the reverse way, i.e. *inverting the resolution*.

Both ways are used in learning by inverting resolution and they can be called *absorption* and *identification* operators respectively. In these operators, the clause $A$ has the role of an example, whereas one of the premises (the given one) represents the background knowledge known in advance. Applying these operators and some other operators described below, one will be able to learn (to extend) theories represented in the form of Horn clauses.

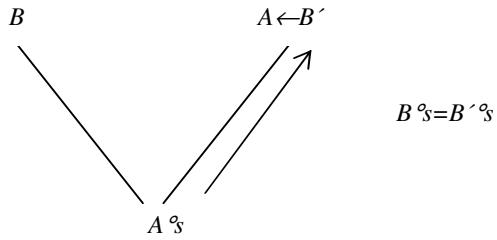



**Figure 3.8.** Resolution with unification

In the case of the predicate logic, the resolution step involves unification – using the most general unifier (mgu) $s$ for literals $B$ and $B'$. This resolution step is shown in Fig. 3.8. In this case, reconstruction of one premise from the resolvent and another premise is no more unique, because the substitutions needed for unification can be chosen in several ways. This leads us to search that can be guided by the heuristics that give preferences to simplicity (Muggleton, Buntine [37]). As the inverse resolution in general is not unique, then an oracle can be used – it will be asked whether the hypotheses suggested are correct or not.

In the predicate logic, we can obtain more information for reconstructing a clause by considering more than one resolution step at a time. Fig. 3.9 shows an inverted resolution operator called *intra-construction* that combines two resolution steps, each of which resolves on the same literal $L$.

Given non-empty conclusions, the configuration in Fig. 3.9 can be used for reconstructing all premises. This is especially interesting, because it enables one to invent new (!) predicates. Indeed, the literal *L* disappears in resolvents *A<-B1* and *A<-B2*, but terms in the literals *A, B1* and *B2* bear some information for reconstructing the literal *L*.

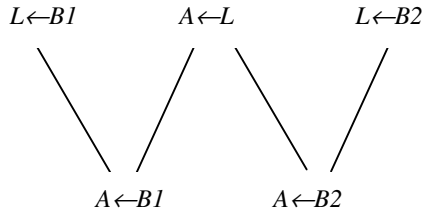



**Figure 3.9.** Combining inverted resolution steps

The resolution steps that resolve to empty clauses can be used for reconstructing the premise *L* as the least general generalization of literals *L1* and *L2* – one from each resolution step. This is called *truncation*. This is rather similar to what we did in the case of specific to general concept learning. Fig. 3.10 summarizes the inverse resolution operators in a slightly more general form.

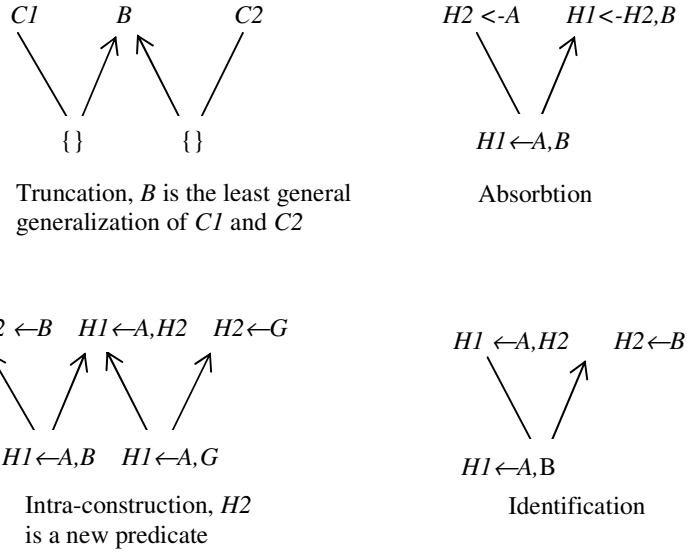



**Figure 3.10.** Inverse resolution operators

Let us have a closer look at the inverted resolution with the clauses containing predicates. We shall denote a resolution step that gives the resolvent *C* from the premises *C1* and *C2* by *C=C1*C2*. The respective inverse resolution step that gives *C2* from *C* and *C1* will be denoted by *C2=C/C1*. We have to specify places where a subterm occurs in a term. This will enable us to distiguish its each occurence in a term. A place is a tuple of

integers $(a_1, a_2,..., a_k)$ where $a_i$ is the number of the position of the subterm of the *i-th* level (counted from left to right) where the described subterm occurs. For example, for the term

*likes(John, brother(John))*

the place of the first occurence of *John* is *(1)* and the place of its second occurence is *(2,1)*.

We define an inverse substitution $s^{-1}$ of a substitution $s=((v_1,t_1),..., (v_k,t_k))$ and terms $t$ and $t'=t \circ s$ as the following tuple of triplets

$$s^{-1}=((t_1,(p_{11},..., p_{1r}),v_1),..., (t_k,(p_{k1},..., p_{kq}),v_k)),$$

where for each pair $(v_i,t_i)$ of the substitution there is a triplet $(t_i,(p_{i1},...,p_{im}), v_i)$ showing the places in $t'$ where the term $t_i$ has been substituted instead of the variable $v_i$. We use also the denotation $t=t' \circ s^{-1}$ in this case. Example: the inverse substitutuion for the substitution *((x,John))* and the terms *t=likes(x,brother(x))* and *t'=likes(John,brother(John))* is *((John,((1),(2,1)),x))*.

Having a substitution $s=((v_1,t_1),..., (v_k,t_k))$ and terms $t$ and $t'=t \circ s$ such that $t$ and $t'$ don't have common variables, and all substitutable variables $v_1,..., v_k$ of $s$ occur in the term $t$, we say that the substitution $s$ is a *s-difference* of the terms $t$ and $t'$, and we denote this by $s=t-_S t'$. When a s-difference of terms $t$ and $t'$ exists, it is uniquely determined by the following equalities:

$t-_S t'=(v,t')$, if $t$ is a variable $v$, and

$f(r_1,..., r_n) -_S f(q_1,..., q_n)= ((r_1 -_S q_1),..., (r_n -_S q_n))$ in general case.

When $t-_S t'$ is determined, we say that the term $t$ is a generalization of the term $t'$. For example, *plus(x,y)* is a generalisation of *plus(3,4)*, because $plus(x,y)-_S plus(3,4)$ exists (it equals to *((x,3),(y,4))* ).

Now we are prepared to consider an inverse resolution step *C2=C/C1* where *C1* and *C2* don't have common variables. Let us denote by *L1* and *L2* the resolvable literals in *C1* and *C2*, and by *s* their unifier, i.e. we have (where – denotes difference of sets)

$$C=(C1-\{L1\}) \circ s \ (C2-\{L2\}) \circ s.$$

As the clauses *C1* and *C2* do not have common variables, we can represent the substitution *s* as a composition of substitutions *s1* and *s2* where *s1* changes only *C1*, and *s2* changes only *C2*:

$$C=(C1-\{L1\}) \circ s1 \cup (C2-\{L2\}) \circ s2.$$

From the resolution condition follows that $L1 \circ s1=L2 \circ s2$, and introducing the inverse substitution of *s2, L1* and *L2*, we get $L2=L1 \circ s1 \circ s2^{-1}$, and

$$C2=(C-(C1-\{L1\}) \circ s1 \circ s2^{-1} \cup \{L1\} \circ s1 \circ s2^{-1}=$$
$$=((C-(C1-\{L1\}) \circ s1 \cup \{L1\} \circ s1) \circ s2^{-1}.$$

We have three unknown entities in the last equation. They are *L1, s1* and $s2^{-1}$. In the case of unary resolution step *C1=L1*, and we get

$$C2=(C \cup \{L1\} \circ s1) \circ s2^{-1} .$$

We know as well that $s=s1 \circ s2$ is the most general unifier of *L1* and *L2*. Finally, this determines a search space of acceptable size for finding *s1* and $s2^{-1}$.

## 3.3. Massively parallel learning in genetic algorithms

Genetic algorithms belong to learning algorithms that can very much gain from the high-performance and massively parallel hardware, because most of computations can be done in parallel. The efficiency of a particular genetic algorithm depends on its tailoring to a particular domain where it will be applied. A genetic algorithm can be explained in the biological terms. A goal of the algorithm is to develop by evolution objects that are good in some sense. The objects are represented by their codes that are called *chromosomes*. A *genetic algorithm* is presented by means of the following four components:

1. *method of encoding* of objects on chromosomes;
2. *fitness function* – evaluation function of chromosomes;
3. *method of initialization* of chromosomes;
4. *mechanism of reproduction* that consists of selection of chromosomes for reproduction, *crossover* and *mutation* mechanisms.

The algorithm starts with initializing a gene pool in the form of a sufficiently large set of chromosomes. Thereafter, the algorithm creates new and new generations of objects represented by their chromosomes, using a reproduction mechanism specifically tailored for a particular problem. As a result of evolution, the gene pool improves and better individuals appear. The algorithm stops, when a sufficiently good individual is created or the time of evolution runs out. Obviously, interference by a user and some interactive guidance to the algorithm are possible.

For computational purposes, chromosomes are very often represented as binary vectors. Because all functions operate on these vectors, the development of suitable encoding on chromosomes is both difficult and crucial for the successful application of the algorithm. There are a number of impressive examples of application of genetic algorithms, for instance, in solving some scheduling problems, multimodal and discrete optimization etc.

We present here a simple example problem taken from an old "Byte" magazine that illustrates a design of a genetic algorithm. The goal is to find an expression for computing the value of *x* from the equations

$$ax + by = c$$
$$dx + ey = f, \text{ where } ae - bd = 1.$$

As we can check, the correct answer is *ec - bf*. The fitness function can be found as a specialization of the following more general fitness function applicable in many cases where numeric values of discrepancies (*currentvalue - correctvalue*) can be computed:

$$fitness = \frac{1}{1 + \Sigma (currentvalue - correctvalue)^2}.$$

This gives us the following fitness function for our example:

$$fitness = \frac{1}{1 + (ax + by - c)^2 + (dx + ey - f)^2}.$$

In order to choose the encoding of chromosomes, we must understand the essence of the objects (the phenotypes) represented by the chromosomes. In our example, the objects are computations that we can represent by binary trees. Fig. 3.11 shows two possible chromosomes (neither of them represents the solution of our problem). The crossover operator applied to two chromosomes cuts off a branch of both chromosomes and swaps the branches between the chromosomes. As the result, we get new chromosomes shown in Fig. 3.12. Cutting points of chromosomes are chosen randomly. This offers many possibilities to get new chromosomes by applying the crossover operator.

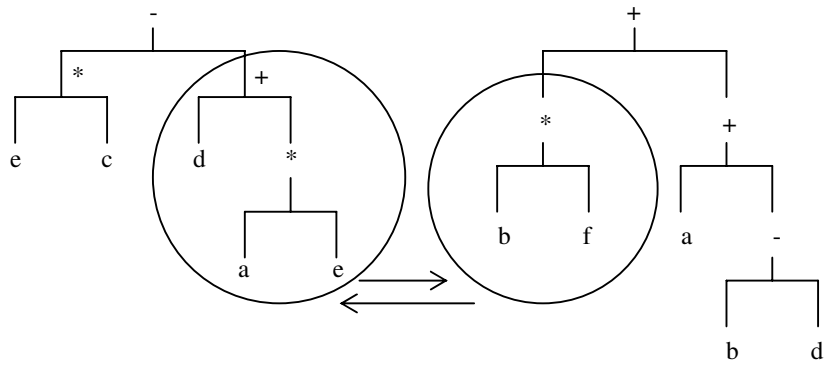



**Figure 3.11.** Crossover = cut and swap

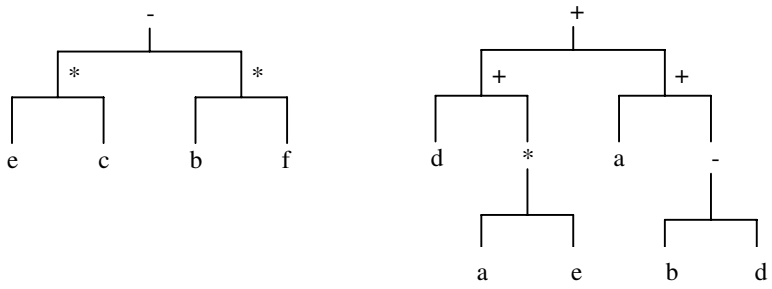



**Figure 3.12.** Results of crossover

To get new chromosomes a mutation operator must be applied. This operator changes randomly a very small part of a chromosome (often a single bit). In the present example, it is useful to change one node of a tree, preserving the shape of the tree. Having

designed the operators, we can apply the general algorithm shown in A.3.10. This algorithm contains, in addition, a normalization operator that in our case can be omitted (substituted by a dummy operator).

There is no difference whether we call representation of individuals in a genetic algorithm chromosomes or genes, because here the similarity with biology ends. We need the following constants in the genetic algorithm:

*maxGenes* – maximal number of genes used;
*maxGenerations* – maximal number of generations;
*mutationProbability* – probability of mutations.

We use also the following notations:

*genePool[]* –array of genes;
*fitnessValue[]* – array of fitness values of genes;
*createRandomGene()* – generator of a gene;
*evaluateGene(x)* – function that calculates a fitness value of a gene *x*;
*normalize(m)* – procedure of normalization of fitness values of the gene pool presented as an array *m;*
*selectPairs()* – function for selecting a pair for crossover;
*crossover()* – crossover operator;
*mutate(p)* – mutation operator that mutates genes with probability *p*.

**A.3.10:**

```
for i= 1 to maxGenes do
        genePool[i]=createRandomGene();
od;
for generation=1 to maxGenerations do
        for i=1 to maxGenes do
                fitnessValue[i]=evaluateGene(GenePool[i])
        od;
        normalize(FitnessValue);
        selectPairs();
        crossover();
        mutate(mutationProbability);
od
```

Performance of a genetic algorithm depends very much on the fitness function. Fitness function should guide the search to the global optimum or at least to a good solution. In particular, a fitness function that tells only good or not good about an individual will not lead to a solution of the problem.

## 3.4. Learning in neural nets

Neural nets provide another form of massively parallel learning. They are well suited for learning pattern recognition. They can be implemented either in hardware or in the form of programs. A simple way to describe a *neural net* is to represent it as a graph. Each node of the graph has an associated variable called *state* and a constant called *threshold*. Each arc

of the graph has an associated numeric value called *weight*. Behavior of a neural net is determined by its structure and transfer functions for nodes that compute new values of states from inputs of the nodes. A common transfer function for computing a new state is of the form

$$x_j = f(\Sigma w_{ij} * x_i - t_j)$$

where summation is done over all incoming arcs of the node $j$ at which the new state $x_j$ is being computed; $w_{ij}$ are weights of these arcs and $x_i$ are current states of the neighbouring nodes; $t_j$ is a threshold of the node $j$. The most typical transfer functions called *hard limiter*, *treshold logic* and *sigmoid* are shown in Fig. 3.13.

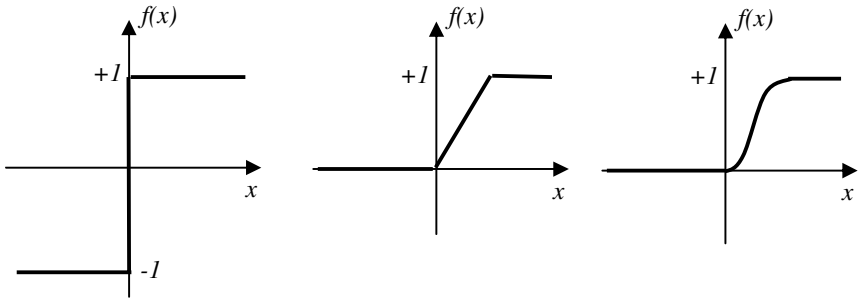



**Figure. 3.13.** Transfer functions

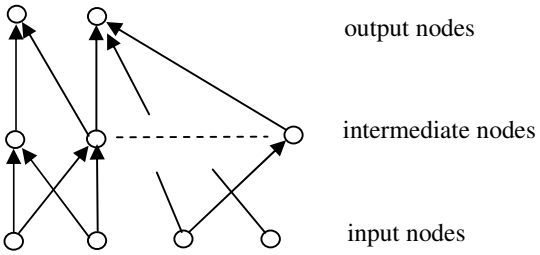



**Figure 3.14.** Example of a forward-pass layered neural net

Neural nets can be classified on the basis of their general structure into the following classes:

- *Forward-pass neural net* is an acyclic graph. Its nodes can be classified as input, output and internal nodes. Input nodes do not have incoming arcs, output nodes do not have outgoing arcs and internal nodes possess both kinds of incident arcs.
- *Layered neural net* (*n*-layered net) net is a net where nodes can be divided into *n* layers so that each layer contains only nodes of one type, reachable from an input layer by a path of one and the same length. Each node in such a graph belongs exactly to one layer; *n*-layered net is strongly connected, if each node in the *i*-th layer is connected to all nodes of the *(i+1)*-st layer, *i=1,2, . . . ,n-1*. States of output nodes of a forward-pass layered net can

be interpreted as decisions made on the basis of the states of the input nodes. Fig. 3.14 shows an example of a forward-pass layered net.

Learning in a layered net can be performed by means of *back-propagation*. In this case, the states taken by output nodes are evaluated and credit or blame is assigned to each output node. The evaluations are propagated back to other layers. The weights of arcs that supported right decisions (of the arcs entering the nodes which received credit) are increased and the weights of arcs that supported wrong decisions are decreased. This method is applicable in nets with small number of layers (2 or 3). In the case of greater number of layers, the credit and blame of output nodes cannot be used for changing weights of layers closer to input.

### 3.4.1. Perceptrons

Well-known representatives of layered nets are *perceptrons*. Examples of perceptrons with two inputs are shown in Fig. 3.15. They are applicable for classification of points of a plane (of a 2-dimensional space). In the case of $n$ inputs the classification occurs in a $n$-dimensional Eucleidian space. The simplest is a *single-layer perceptron* that can classify points into two classes, if the points are separable by a linear boundary (by a hyperplane). Indeed, considering hard limiter as a transfer function of the perceptron, we get that the output of a single-layer perceptron will be –1, if $\Sigma w_{ij} * x_i < t_j$ and it will be +1, if $\Sigma w_{ij} * x_i > t_j$.

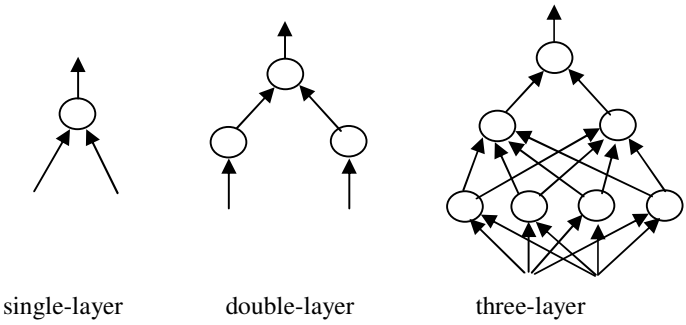



**Figure 3.15.** Perceptrons with 2 inputs

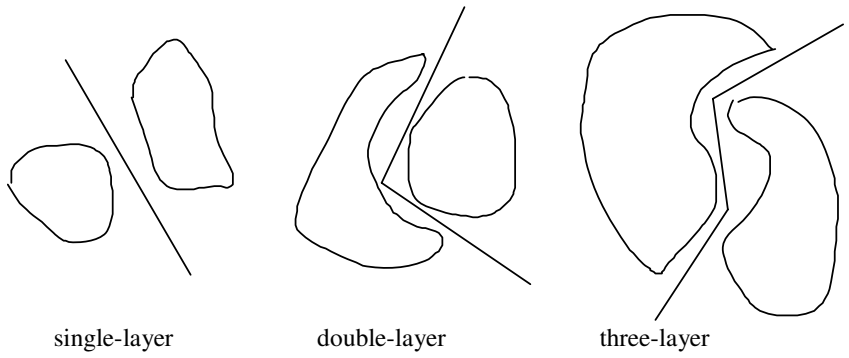



**Figure 3.16.** Examples of regions separable by pereptrons with two inputs

A *double-layer perceptron* can separate any convex region. A three-layer perceptron can theoretically separate any complex region. Examples of regions separable in two-dimensional space by different perceptrons are shown in Fig. 3.16. For a single-layer perceptron, the boundary is a straight line, for a double-layered perceptron it is a polyline surrounding a convex area, and for a three-layered perceptron it can be any polyline.

Learning in a multilayer perceptron can be performed by back-propagation algorithm that minimizes the mean square error between the actual output and the desired output. It works in the case of continuous differentiable transfer functions. When learning in a preceptron, it is reasonable to use a sigmoid transfer function $f(x)$, where

$f(x)=1/(1+e^{-(x-t)})$, $x$ is sum of weighted inputs and $t$ is offset.

Let us assume here that we have a three-layer perceptron that computes binary output where only one component is *1*, other are *0*. When learning begins, all weights and node offsets $t$ are initialized to small random numbers, if no better values are known. Learning is done step by step, presenting at each step an input and the desired output that corresponds to the input. A learning step begins with taking an input $x_1,\ldots, x_n$ and calculating the output $y_1,..,y_m$. Thereafter new weights $w'_{ij}$ are calculated for output nodes $j=1,\ldots,m$, using the formula

$$w'_{ij} = w_{ij} + a*\delta_j * x_i ,$$

where $a$ is gain (a sufficiently small number that may decrease in time) and $\delta_j$ is error for the node $j$ computed as

$$\delta_j = y_j*(1- y_j)*(d_j - y_j),$$

where $d_j$ is the desired output of node $j$ and $y_j$ is computed output of node $j$. This is followed by calculation of new weights for the hidden layer. The formula for calculating new weights for the hidden layer is again

$$w'_{ij} = w_{ij} + a*\delta_j * x_i ,$$

but the error $\delta_j$ is calculated now as follows:

$$\delta_j = x_j*(1- x_j)*\sum_s \delta_s * w_{js} ,$$

where $s$ is over all output nodes – it takes into the account all errors of the output layer.

Learning in perceptrons with more than three layers is quite difficult, because it converges slowly, therefore perceptrons with more than three layers are seldom used.

## 3.4.2. Hopfield nets

There are interesting and rather regular nets with feedback, called *Hopfield nets*. They are very simple classifiers, mostly used with binary inputs, for instance, for classifying graphical patterns, or as content-addressable memories. Signals in a Hopfield net can take values +1 and –1. Transfer functions of the nodes are hard limiters. A structure of the Hopfield net is shown in Fig. 3.17. Here $x_1, x_2,..., x_n$ are input signals, $x_1', \ldots x_n'$ are output signals which are equal to the values of states of the nodes. A tuple of input signals

represents an exemplar that must be classified. Also a class is represented by a tuple of $n$ elements, each element corresponding to one of input nodes. We shall denote by $x_{is}$ the $i$-th component of the representation of the class $s$. Weights of the connections are taken as follows:

$w_{ij}=\Sigma x_{is}*x_{js}$, if $i\neq j$,
$w_{ii}=0$ .

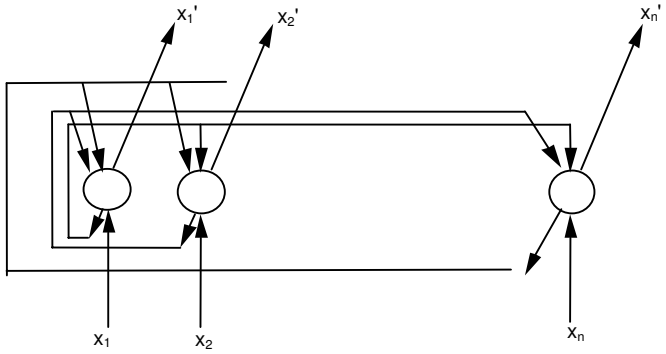



**Figure 3.17.** Hopfield net

This net contains feedback for each node. After getting input signals, it starts operating, computing new states (i.e. new values of output) from the given values on input and previous states which are initially taken equal to input values. This process is continued until outputs become unchanged. The outputs represent then the exemplar pattern that best matches the given input. A Hopfield net performs well, if the number of classes it should recognize is less than 0.15 times the number of its nodes. This condition is fulfilled, for example, for recognizing noisy patterns of ten digits 0,1,..,9 represented by 70 or more pixels. The number of connections between the nodes of this network is quite large, it is about 5000 for recognizing the ten digits.

## 3.4.3. Hamming nets

Another net for solving classification problems of binary vectors is *Hamming net*, shown in Fig. 3.18. This net computes Hamming distances between a given exemplar (which has to be classified) and representatives of all classes, and selects the class with minimal distance from the given example. *Hamming distance* between two binary vectors is the number of bits in these vectors that do not coincide in them. Classification problems that are solved by the Hamming nets occur when binary fixed-length messages are sent through a memoriless noisy channel.

Hamming net consists of two layers, see Fig. 3.18. The first layer calculates Hamming distances, and the second layer, that looks similar to a Hopfield net, picks up the class with minimal distance from the given input sample. Every node of the second layer corresponds to one of the classes of patterns to be recognized.

Let us give net parameters in a net, where $n$ is the number of inputs, $m$ is the number of outputs (i.e. classes). Here $w_{ij}$ and $w'_{ij}$ are again connection weights from the node $i$ to

the node $j$; $t$ and $t'$ are thresholds. The transfer function in the nodes of both layers is the threshold function from Fig. 3.13. The parameters for the lower subnet are

$w_{ij}=x_{is}/2$
$t=n/2.$

This guarantees that the output values of the nodes of this layer will be equal to $n$-$h$, where $h$ is the Hamming distance to the respective exemplar pattern. Indeed, the output of the lower level node for the most incorrect code will be 0, and changing an incorrect signal $x_{is}$ to a correct one adds 1 to the sum, so that this gives $n$ for the correct code. The highest output will belong to the node of the class that matches best. Parameters for the upper subnet are

$w'_{ij}$=1, if $i=j$ and
$w'_{ij}$= - $e$, where $0<e<1/m$, if $i\neq j$,
$t'=0.$

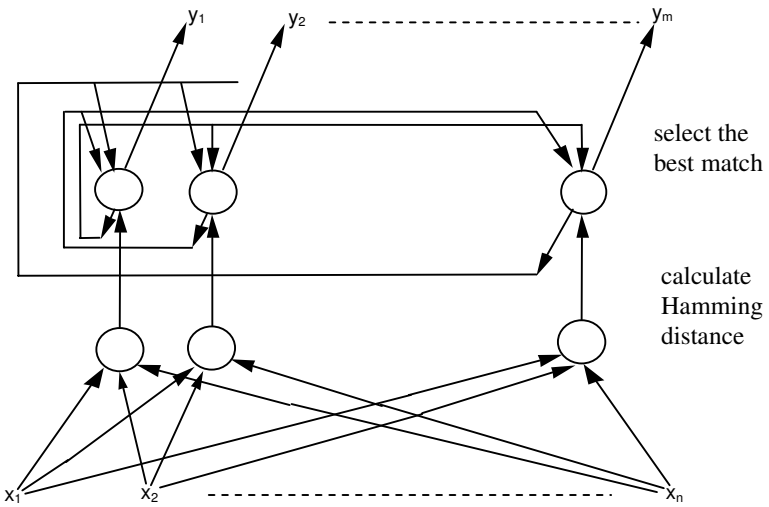



**Figure 3.18.** Hamming net

The weights $w'_{ij}$ between different nodes are inhibitory – they are with a small negative value $-e$, and $w'_{ii}$ for each node $i$ gives a positive feedback. This gives a net that selects the node with maximal value of input. This net is called also *maxnet*.

After getting input signals, the first layer of the net computes Hamming distances that are then passed to the second layer, to the maxnet. This layer computes new states (i.e. new values of outputs of nodes) from the given values on its input and previous states which are initially taken equal to its input values. This process is continued until outputs become unchanged and the output of only one node is above zero. The node with this output represents then the class that best matches the given input. This process converges approximately in 10 iterations.

The Hamming net implements a *minimum error classifier* when bit errors are random and independent. The Hamming net requires fewer connections than the Hopfield net for the same number of inputs.

## 3.4.4. Comparator

Now we show how a hybrid neural net can handle analog signals. Fig. 3.19 shows a *comparator subnet* that selects the maximum of two nonnegative analog inputs *x0, x1*. Output *z* is the maximum value, $y_0$ and $y_1$ indicate which input is maximum, filled nodes are hard limiters, light nodes are threshold logic nodes, all thresholds are *0*, weights are shown on arcs.

This net works as follows. Nodes of the first layer compare the inputs, and the greater input (let it be $x_0$) produces the value $x_0 - x_1$, another input produces *0*. Hard limiters of the second layer indicate which input is larger. Value of *z* is computed by the formula

$$0.5*x_0 + 0.5*(x_0 - x_1) + 0.5*x_1 = x_0.$$

In the case when $x_1 > x_0$ the roles of $x_0$ and $x_1$ are exchanged:

$$0.5*x_1 + 0.5*(x_1 - x_0) + 0.5*x_0 = x_1.$$

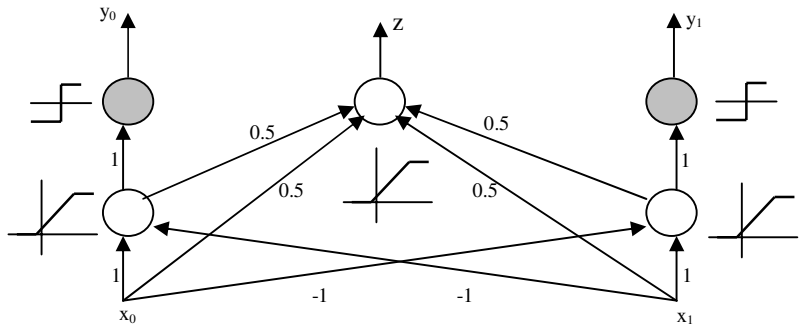



**Figure 3.19.** Comparator network

Combining several of these simple comparator nets, one builds comparators for more inputs (4, 8 etc., approximately $log_2 n$ layers for *n* inputs). Compared to *maxnet* from the previous section, this net not only detects the maximal input, but produces also the maximal value on the output.

## 3.4.5. Carpenter-Grossberg classifier

Hopfield and Hamming nets belong to the binary nets trained with supervision. An example of a binary net trained without supervision is the Carpenter-Grossberg classifier. Fig. 3.20 shows a *Carpenter-Grossberg net* with three inputs $x_0$, $x_1$, $x_2$. It forms clusters analogously to simple *sequential leader clustering algorithm*. This algorithm selects the first input as the exemplar of the first cluster. Thereafter the following procedure is applied: if the distance between the next input and a leader is less than a given threshold, the input

follows the leader, i.e. is classified as the leader has been, otherwise it is the exemplar of a new cluster.

This algorithm is implemented in the form of a two-layered net. This net includes much feedback (see double arrows in Fig. 3.20) and its behavior can be precisely described by nonlinear differential equations.

Let us denote by $t_{ij}$ the top-down connection weight and by $b_{ij}$ the bottom-up connection weight between input node $i$ and output node $j$ of a Carpenter-Grossberg net with $n$ inputs $x_1,\ldots, x_n$ and $m$ outputs $y_0,..,y_{m-1}$. After learning, these weights define the exemplar specified by output node, taking into account the vigilance $v$ of nodes, $0<v<1$. The vigilance is a threshold that shows how close an input must be to an exemplar that it should match. The value of $v$ close to $1$ means that an input must be close to the learnt pattern in order to be classified respectively. Operation of a Carpenter-Grossberg net requires additional control by switching the connections of nodes on and off during learning, therefore a real Carpenter-Grossberg net is more complicated than shown in Fig. 3.20. We will explain the operation of this net in several stages.

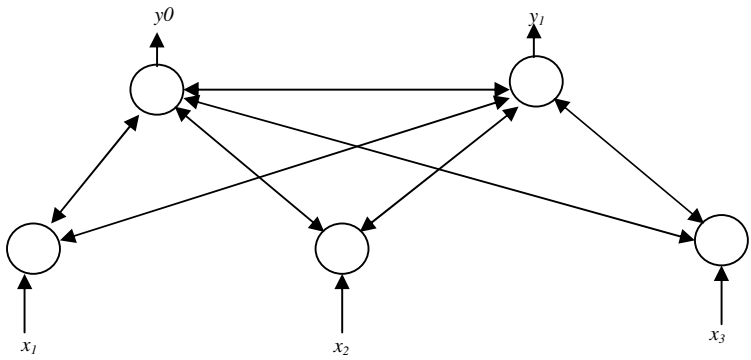



**Figure 3.20.** Carpenter-Grossberg net for three binary inputs and two classes

1. Initializaton:

$t_{ij}=1$
$b_{ij}=1/n$
set $v$ to some number $0<v<1$.

2. For a new input a) compute matching scores $w_j$, b) select the best matching exemplar $k$ with largest $w_k$, and c) perform vigilance test. This is done as shown below.

a) Scores $y_j$ are values of outputs of nodes of the second layer of Carpenter-Grossberg net calculated as follows:

$$y_j=\sum_{i=1}^{n} b_{ij}*x_i$$

b) The best matching exemplar is selected by finding $k$ where

$$y_k = \max_j y_j$$

This is done in the second layer by lateral inhibition, using connections between the neighboring nodes.

c) The vigilance test is performed by dividing the dot product $p$ of input and best matching exemplar (ie the number of bits in common) by the number of bits $q$ equal to *1* in the input, and comparing this ratio with the vigilance $v$. The test is passed, if $p/q>v$, where

$$q=\sum_{i=1}^{n} x_i \quad \text{and} \quad p=\sum_{i=1}^{n} t_{ij}*x_i \ .$$

The computations performed by a Carpenter-Grossberg net and desribed above can be explained also by the algorithm A.3.11 that uses the same notations as above:

**A.3.11:**

```
w_k =0;
p=0;
q=0;
for i=1 to n do
        q=q+ x_i
od
for j=0 to m-1 do
        w_j =0;
        for i=1 to n do
                w_j = w_j + b_ij * x_i;
                p=p+ t_ij * x_i
        od;
        if w_k< w_j and w_j >p/q then
                w_k= w_j
od
```

If the vigilance test fails for the best matching exemplar $k$, then the best matching node $k$ is excluded from processing the given input, and the step 2 is repeated. If the vigilance test succeeds, then the given input is used for learning by computing new connection weights $t'_{ij}$ and $b'_{ij}$ as follows:

$$t'_{ij} = t_{ij} * x_i$$

$$b'_{ij} = t_{ij} * x_i \ /(0.5+p), \quad \text{where} \quad p =\sum_{i=1}^{n} t_{ij}*x_i \ .$$

If the vigilance test fails on some input, then it is possible to add the input as a new exemplar that is learnt. This requires, however, one new node and $2n$ new connections in the net. More about this net can be found in [24].

## 3.4.6. Kohonen's feature maps

There are interesting nets with continuous signals trained without supervision called *Kohonen's feature maps*. They mimic organizing principle of sensory pathways in the brain by orderly placement of neurons and bindings between them. They consist of two layers of nodes: input layer and the second (topological) layer. The nodes of the second layer are extensively connected by many local connections, built up by means of unsupervised learning.

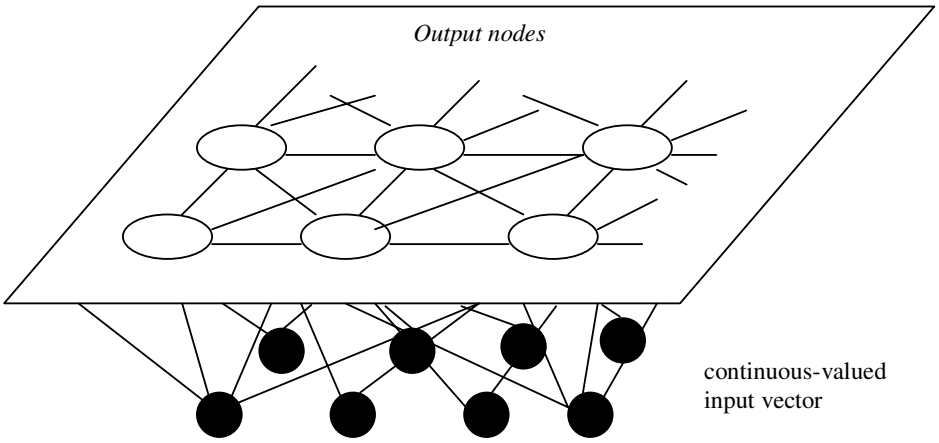



**Figure 3.21.** Kohonen's feature map

Learning in a Kohonen's feature map (Fig. 3.21) goes on as follows. Continuous-valued input vectors are presented to the first layer of the nodes without specifying the desired output. After sufficiently long learning, weights of connections will represent a topology of a two-dimensional space where closeness of points reflects the similarity of presented samples. The learning algorithm is as follows:

1. Initialize weights to small random numbers and set initial radius of neighborhood of nodes.
2. Get an input vector $x_1, \ldots, x_n$.
3. Compute distance $d_j$ of the input vector to each output node:
   $dj = \Sigma (x_i - w_{ij})^2$
4. Select output node $s$ with minimal distance $d_s$.
5. Update weights for the node $s$ and all nodes in its neighborhood:
   $w_{ij}' = w_{ij} + h^* (x_i - w_{ij})$,
   where $h<1$ is a gain that decreases in time.
   Repeat steps 2 - 5.

## 3.4.7. Bayesian networks

Bayesian networks use the conditional probability formula that binds the probability $P(e,H)$ of occurrence of evidence $e$ together with hypothesis $H$ it supports, and conditional

probabilities *P(H|e), P(e|H)* of hypothesis and evidence with unconditional probabilities *P(H), P(e)* of hypothesis and evidence:

$$P(e,H)=P(H|e)P(e) = P(e|H)P(H).$$

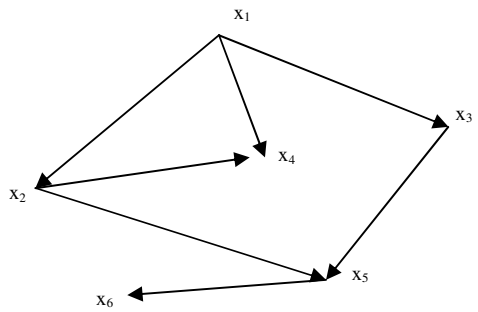



**Figure 3.22.** Bayesian network

*Bayesian network* is a graph whose nodes are variables denoting occurrence of events, arcs express causal dependence of events. Each node *x* has conditional probabilities for every possible combination of events influencing the node, i.e. for every collection of events in nodes of *pred(x)* immediately preceding the node *x* in the graph. Figure 3.22 shows an example of a Bayesian network with six nodes.

The joint probability assessment for all nodes $x_1,\ldots,x_n$ is

$$P(x_1,\ldots,x_n)= P(x_1|pred(x_1))*\ldots \ldots *P(x_n|pred(x_n)),$$

and it expresses a joint-probability model that supports the assessed event combination. For the present example it is as follows:

$$P(x_1,\ldots,x_6) = P(x_6|x_5)*P(x_5|x_2,x_3)*P(x_4| x_1, x_2)*P(x_3| x_1)*P(x_2| x_1)*P(x_1).$$

A Bayesian network can be used for diagnosis/classification: given some events, the probablities of events depending on the given ones can be predicted. To construct a bayesian network, one needs to determine its structure (topology) and to find conditional probabilities for each dependency. A Bayesian network can be specified by an expert. But the structure and probabilities of a Bayesian network can be learnt as well.

It may be difficult to use the joint probability formula of the whole network, because it requires much data and computations. Problems can be formulated on parts of the network. For example, let us assume that the events $x_1$, $x_2$ and $x_4$ in the network presented in Fig. 3.22 have the meaning "rain", "sprinkler on" and "grass wet". Let the conditional probabilities of "sprinkler on" and "grass wet" be given by tables 3.1 and 3.2.

It is obviously easy to find the probability of "grass wet", if the states of sprinkler and rain are known. But even more complicated problems can be solved. Let the probability of rain be 0.2. Now we can solve several problems. For instance, we can ask: "What is the probability that it is raining, if we know that the grass is wet?" This situation is specified by the formula

$$P(x_1|x_4) = P(x_1,x_4)/P(x_4).$$

**Table 3.1**
Conditional probability of sprinkler

| | then sprinkler on | |
|---|---|---|
| if rain | *true* | *false* |
| *false* | 0.4 | 0.6 |
| *true* | 0.01 | 0.99 |

**Table 3.2**
Conditional probability of wet grass

| | | then grass is wet | |
|---|---|---|---|
| if sprinkler on and rain | | *true* | *false* |
| *false* | *false* | 0.0 | 1.0 |
| *false* | *true* | 0.8 | 0.2 |
| *true* | *false* | 0.9 | 0.1 |
| *true* | *true* | 0.99 | 0.01 |

To find the probability $P(x_1,x_4)$, one has to take a sum over the cases with "sprinkler on" *true* and *false*:

$$P(x_1,x_4) = P(x_1,x_2,x_4) + P(x_1,\text{not } x_2,x_4) = 0.1584 + 0.00198.$$

To find $P(x_4)$, one has to take the sum over four cases for $x_1$, $x_2$ both *true* and *false*:

$$P(x_4) = P(\text{not } x_1,\text{not } x_2,x_4) + P(\text{not } x_1,x_2,x_4) + P(x_1,x_2,x_4) + P(x_1,\text{not } x_2,x_4) = 0 + 0.288 + 0.1584 + 0.00198,$$

and the answer will be that the probability of rain when grass is wet is

$$P(x1|x4) = (0.1584 + 0.00198)/(0 + 0.288 + 0.1584 + 0.00198) \cong 0.36.$$

A Bayesian network in Fig. 3.23 describes a situation, where possible events are related to alarm in a house, and to calling that depends on the alarm. The events are *theft, earthquake, alarm*, *T calls* (Tom callas) and *A calls* (Ann calls). Unconditional probabilities of theft and earthquake are known (0.001 and 0.002 respectively). Conditional probabilities of alarm and calling are given in tables at the respective nodes of the network in Fig. 3.23.

A number of questions can be answered already on this simple network. For example, if one wants to know the probability that Tom will call, if there is theft and no earthquake, then one can get from the table of alarm the respective probability 0.94, and from the table of Tom calling the probability 0.9. The answer will be 0.9*0.94=0.85. A slightly more complicated problem is to find a probability that theft has occurred when Peter is calling. This is called a diagnostic problem (from consequences to sources). The answer will be 0.016.

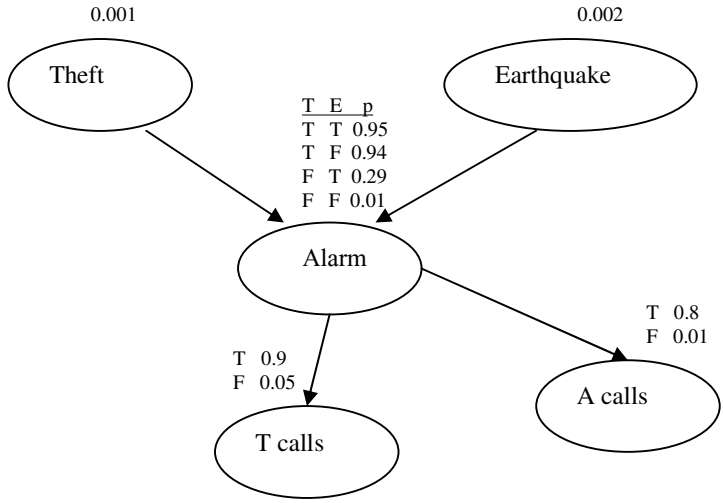



**Figure 3.23**. Bayesian network with given probabilities.

## 3.4.8. Taxonomy of neural nets

Fig. 3.24 shows a taxonomy of neual nets where type of signals (binary or continuous), way of learning (supervised or unsupervised) and number of layers are taken into account.

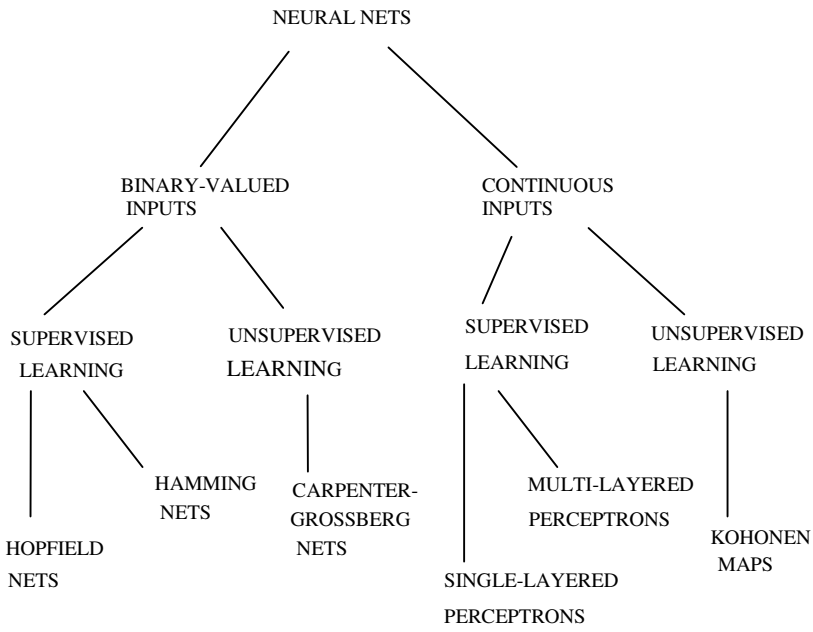



**Figure 3.24.** Taxonomy of neural nets

Hopfield and Hamming nets belong to the nets with binary signals and supervised training. They are easy to use, because parameters of their nodes can be precisely computed from the given representatives of the classes.

Perceptrons are nets with continuous input signals and supervised training. Perceptrons are with a long history, and they are probably the most popular neural nets. Nets with unsupervised training are Carpenter-Grossberg nets and Kohonen's feature maps. Single-layer nets are Hopfield nets and single-layer perceptrons. These nets have restricted capabilities, as it could be expected. We have presented only a small number of neural nets as examples here compared to the great variety of existing neural nets.

## 3.5. Data clustering

There are algorithms of learning which do not require the presence of a tutor. These algorithms of unsupervised learning detect some regularities or laws in data and are based mainly on statistics. A simple case of *data clustering* that is a kind of statistics-based learning is illustrated in Fig. 3.25. In this example, we have to do with objects that are numerical values of a variable $x$, i.e. points in one-dimensional space that form two well-distinguishable sets (clusters), as seen from Fig. 3.24 that shows the frequency $f$ of occurrence of the values of $x$. These clusters can be recognized easily by simple statistical methods. Descriptions of classes (i.e. of concepts) are the average values $x1$ and $x2$ of the objects of the classes together with some measure of dispersion of the values in a cluster.

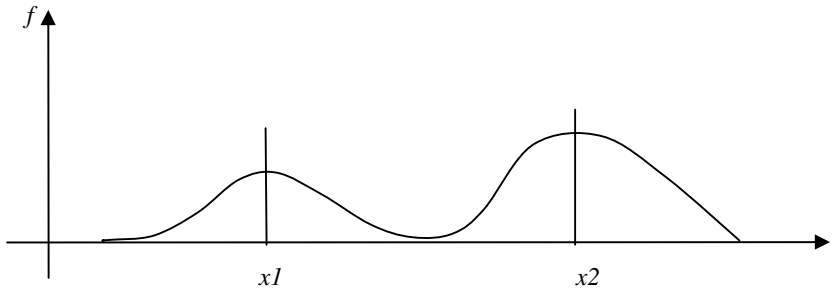



**Figure 3.25.** Example of a clustering problem

There are plenty of clustering algorithms developed for learning without supervision clusters of objects from large sets of objects. Ofen these algorithms do not use much statistics, trying to process the objects sequentially and each object only once. They collect similar objects into a set of similar objects called *cluster*, using a concept of distance between the objects. The distance can be defined in different ways, and it depends on the properties of objects, more precisely – on the properties of a space where the clustered objects occur. Let the objects $a$ and $b$ be presented by $n$ properties $a_1,\ldots,a_n$ and $b_1,\ldots, b_n$ respectively. Here we give some definitions of a distance between the objects.

*Euclidean distance*

$$d = sqrt\ (\sum_i (a_i - b_i)^2),$$

*maximum metric distance*

$d = \max_i abs(a_i - b_i)$,

*city block distance*

$d = \sum_i abs(a_i - b_i)$,

and *Hamming distance*

$d = k$, where $k$ is the number of differing properties ($a_i \neq b_i$) of objects.

## 3.5.1. Sequential leader clustering

Probably the simplest clustering algorithm is *sequential leader clustering algorithm*. It uses the notations

*dist(x,y)* -- a function for calculating a distance between the objects $x$ and $y$, where $y$ may be a cluster as well;
*createCluster(x)* -- a function for creating a cluster that includes one element $x$;
*addToCluster(x,c)* -- a procedure for adding element $x$ to cluster $c$;
*selectFrom(clusters)* – a procedure that selects an element from *clusters*;
*clusters* – set of clusters, each of them represented by an element;
$e$ – a threshold of closeness of exemplar, required for introducing it into a cluster.

The algorithm A.3.12 presents one step of the sequential leader clustering algorithm. It must be repeated for each new exemplar.

**A.3.12:**

```
seqLeader(x,clusters,e) =
    if empty(clusters) then clusters = {createCluster(x)} fi;
    dmin=bigNum;
    cm=selectFrom(clusters);
    for c∈ clusters do
            dd=dist(x,c);
            if dmin>dd & dd≤e then
                    dmin= dist(x,c);
                    cm=c
            fi
    od;
    if dmin<bignum then
            addToCluster(x,c)
    else
            createCluster(x)
    fi
```

## 3.5.2. K-means clustering

A cluster can be represented by its first element, as it is often done in the sequential leader clustering algorithm. But the cluster can be represented also by a more typical

element, or by some constructed element, e.g. by its average element. The element that represents a cluster is called its *centroid*, and it can be recomputed. This is used in a *K-means clustering algorithm*. Besides the notations introduced above, this algorithm uses the following notations:

*ex* – a set of exemplars to be clustered;
*k* – number of clusters, given as an argument;
*clusters* – set of clusters that are sets of similar elements;
*centroid(c)* – centroid of cluster *c*;
*newPos(c)* – recalculates position of centroid of the cluster *c;*
*initialize(k)* – initializes *k* clusters by selecting a centroid for each cluster and constructing a set that includes only the centroid.

**A.3.13**

```
kMeansClustering(x,clusters,k) =
    clusters=initialize(k);
    for x∈ ex do
        dmin=bigNum;
        cm=selectFrom(clusters);
        for c∈ clusters do
                dd=dist(x,centroid(c));
                if dmin>dd then
                        dmin= dist(x,centroid(c));
                        cm=c
                fi
        od;
        addToCluster(x,c);
    od;
    for c∈ clusters do
        newPos(c)
    od;
```

New centroid *centr* is calculated by *newPos(c)* so that its coordinates $centr_i$ are the arithmetic mean for each dimension *i* separately over all the points $x_j$ , *j=1,…, m* in the cluster:

$$centr_i = \sum_{j=1}^{m} x_{ji} / m \, .$$

This can be done by the following small algorithm:

**A.3.14:**

```
newPos(c)=
        result=();
        for i=1 to n do
                m=0;
                sum=0;
                for x∈ c do
                        m=m+1;
                        sum=sum+ x_i
```

```
            od;
            append(result,sum/m))
      od;
      return result
```

It is obvious that the K-means clustering algorithm can work well, if it is possible to guess good seeds for clusters. A simple guess is to place the seeds as far away as possible from each other. This algorithm can be repeated several times, taking already found centroids as the new seeds. This can be repeated until no changes of centroids occur. A drawback of the algorithm is the amount of work required for adjusting the centroids. Many improvements of basic clustering algorithms exist that make them applicable for large amounts of data as well.

## 3.6. Specific learning algorithms

### 3.6.1. Learning decision trees from examples

A decision tree is a simple form of knowledge specifically suitable for representing knowledge for decision-making. It is a tree with nodes marked by attributes, attribute values or decisions. Its root is always marked by an attribute. Each path starting from the root passes through a node with a value of the attribute of the root and thereafter alternatingly through the nodes marked by attributes and their values. It ends with a node marked by a decision drawn from the values of the attributes met along the path. Another form of a decision tree has values of attributes associated with arcs.

Fig. 3.26 shows a decision tree with attributes describing weather conditions. Possible decisions are divided into two classes denoted by + and - which can represent, for instance, automatically and manually controlled landing of a vehicle. This is a part of a decision tree for the sample data given in Table 3.3.

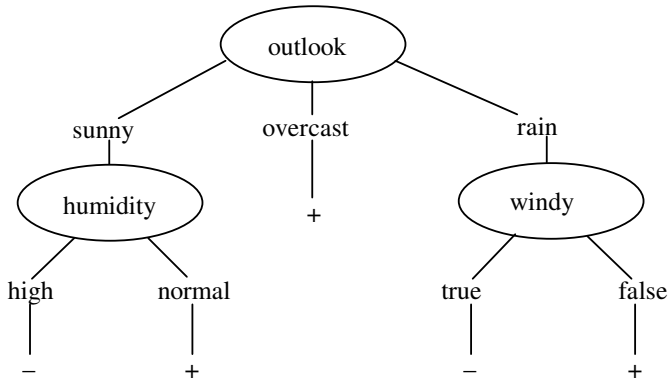



**Figure 3.26.** Decision tree

As we can see from Fig. 3.26, not all attributes are needed for making some decisions. The number of steps needed for making a decision depends on the order in which the attributes appear in the decision tree. We are interested in building a decision tree with minimal number of nodes from a given set of examples. This can be done by means of an algorithm called ID3 (Quinlan [42]) that we shall explain here.

The algorithm is based on the observation that a path from the root to a decision is in average the shortest, if the most discriminating attribute is tested at each step. The most discriminating attribute can be defined in precise terms as the attribute for which the fixing its value changes the enthropy of possible decisions at most. Let $w_j$ be the frequency of the $j$-th decision in a set of examples $x$. Then the entropy of the set is

$$E(x)= - \Sigma w_j * lg(w_j).$$

**Table 3.3.**
Sample data for ID3

| Outlook | Temperature | Humidity | Windy | Class |
|---|---|---|---|---|
| Sunny | Hot | High | False | - |
| Sunny | Hot | High | True | - |
| Overcast | Hot | High | False | + |
| Rain | Mild | High | False | + |
| Rain | Cool | Normal | False | + |
| Rain | Cool | Normal | True | - |
| Overcast | Cool | Normal | True | + |
| Sunny | Mild | High | False | - |
| Sunny | Cool | Normal | False | + |
| Rain | Mild | Normal | False | + |
| Sunny | Mild | Normal | True | + |
| Overcast | Mild | High | True | + |
| Overcast | Hot | Normal | False | + |
| Rain | Mild | High | True | - |

Let *fix(x,a,v)* denote the set of these elements of $x$ whose value of

$$H(x,a) = \Sigma k_V E(fix(x,a,v)),$$

where $k_V$ is the ratio of examples in $x$ with attribute $a$ that have the value $v$. For instance, for the example data from the Table 3.3 and for the attribute "outlook" attribute $a$ is $v$. The average enthropy that remains in $x$, after the value $a$ has been fixed, is:that has three values "sunny", "overcast" and "rain" we get the enthropies

$$E1=-(3/5 \log(3/5)+2/5\log(2/5))=0.971$$
$$E2=-4/4 \log(4/4)=0$$
$$E3=-(3/5 \log(3/5)+2/5 \log(2/5))=0.971.$$

Their weights are respectively

*k1=5/14*
*k2=4/14*
*k3=5/14*

and the resulting enthropy is

*H = 5/14 *0.971+4/14 *0 +5/14 *0.971 = 0.694.*

For a given set of examples, the most discriminating is the attribute *a* for which *H(a,x)* is minimal, i.e. fixing this attribute decreases the enthropy most. We use the following notations for representing the algorithm ID3:

*p* – pointer to the root of the decision tree being built;
*x* – set of examples;
*E(x)* – enthropy of the set of examples *x*;
*H(x,a)* – average enthropy that remains in *x* after the value of *a* has been fixed;
*atts(x)* – attributes of the set of examples *x*;
*vals(a, x)* – values of the attribute *a* in te set of examples *x*;
*mark(p,d)* – mark node *p* with *d*;
*newsucc(p,v)* – adds a new successor to the node *p*, marks the new arc by *v* and returns pointer *p* to the new node;
*fix(x,a,v)* – derives a new set of examples from the given set of examples *x* by fixing the value *v* of the attribute *a;*
*decision(x)* – decision for the set of examples *x*.

The algorithm ID3 is presented in A.3.15.

**A.3.15:**

```
ID3(x,p)=
        if empty(x) then failure()
        elif E(x)==0 then
                mark(p,decision(x))
        else
                h=bignumber;
        for a ∈ atts(x) do
                if H(x,a) < h then
                        h=H(x,a);
                        am=a
                fi
        od;
        mark(p,am);
        for v ∈ vals(am,x) do
                ID3(fix(x,am,v),newsucc(p,v))
        od
fi
```

### 3.6.2. Learning productions from examples

Knowledge for decision-making can be represented in the form of productions. Also in this case there is an algorithm that extracts the decision-making knowledge from examples. An example is a collection of atomic propositions

*attribute = value*

together with the name of the concept that is represented by the example.

The algorithm called AQ builds rules for one concept at a time using examples of this concept as positive information and examples of other (mutually exclusive) concepts as negative information (as counterexamples). The rules have the form

***if*** *A1∧...∧Ak* ***then*** *concept*

where *A1,...,Ak* are atomic propositions taken from examples.

This learning algorithm builds more complex descriptions of concepts than the symbolic learning algorithms described in Section 3.2. A concept can be represented by several rules, which all together can be considered as a single rule with disjunction of conjunctive conditions:

***if*** *A11∧...∧A1k ∨...∨Am1∧...∧Amn* ***then*** *concept.*

The algorithm should use heuristics for selecting good propositions as well as for choosing the most suitable example to use at each step. The algorithm *AQ(ex,cl)* builds a set of rules from the given set of examples *ex* for the collection of concepts *cl*. We use the following notations:

*pos(ex,c)* – the examples from *ex* which are the instances of the concept *c*;
*neg(ex,c)* – the examples which are instances of other concepts;
*covers(r,e)* – predicate which is true when example *e* satisfies the rule *r;*
*seed* – an initial condition for a rule being built;
*newTests(r,seed,e)* – a function which generates candidate amendments *q* to the rule *r* where *r&q* covers *seed* and does not cover *e*;
*worstElem(star)* – a function that chooses the worst element in the set *star*
*bestIn(star)* – selects the best element in *star*.

Most of the work for *AQ* is done by the following two functions:

*aqrule(seed,neg,c)* – builds a new rule from the initial condition *seed* and negative examples *neg* for the concept *c* (this is just a general to specific concept learning algorithm);
*aqrules(pos,neg,c)* – builds the complete description of the concept *c* from given sets of positive and negative examples *pos* and *neg*.

The functions *AQ, aqrules* and *aqrule* are presented in A.3.16. The function *AQ* just calls *aqrules* for each class and collects the rules in one set. The function *aqrules* begins by initializing the set of rules with one rule. Then it checks for each positive example whether it is covered by some rule. If not, then a new rule is added. Finally, the set of rules is pruned – rules that are covered by some other rule are dropped. Looking at the function

*aqrule* one can see the similarity with the beam search algorithm from the search chapter. The function *aqrule* is a concretization of the beam search for the case of learning from examples. The beam is called here *star*. The function *aqrule* does not perform exhaustive search, hence, the *AQ* algorithm does not necessarily find the best solution.

**A.3.16:**

```
AQ(ex,cl)=
        allrules = { };
        for c∈ c1 do
                allrules=allrules ∪ aqrules(pos(ex,c),neg(ex,c),c)
        od;
        return(allrules)


aqrules(pos,neg,c) =
        rules = {aqrule(selectFrom(pos),neg,c)};
        for e ∈ pos do
                L: {for r ∈ rules do
                        if covers (r,e)
                                then break L
                        fi
                    od;
                    rules = rules ∪ {aqrule(e,neg,c)}
                }
        od;
        prune(rules);
        return(rules)


aqrule(seed,neg,c) =
        star={true};
        for e ∈ neg do
                for r ∈ star do
                        if covers(r,e) then
                             star=star∪{r&q|q∈ newTests(r,seed,e)}\{r}
                        fi;
                        while size(star)>maxstar do
                             star=star\{worstElem(star)}
                        od
                od
        od;
        return("if" bestIn(star) "then"c)
```

## 3.6.3. Discovering regularities in monotonous systems

More complicated methods have been developed for discovering regularities in data and for building description of classes by analyzing the data, i.e. by *data mining*. One of such algorithms will be presented here. This algorithm takes data in the form of a table of

examples and extracts a subset of the most closely related examples from this set. The set of examples must satisfy some special conditions – it must constitute a *weakly monotonous system* [38], [50]. We present the algorithm by solving an example of a clustering problem here.

Let us have the following set of examples, where each column contains values of one attribute and each row represents an example:

1 1 1 1 1
1 1 2 1 1
2 1 1 1 1
3 2 2 1 0
2 1 2 2 0

The goal is to detect regularities in these examples and to classify these examples on the basis of these regularities.

The algorithm prescribes the following actions:

1. Find frequencies of occurrence of values of attributes in columns:

| Attribute value | Column nr. 1 2 3 4 5 |
|---|---|
| 0 | 0 0 0 0 2 |
| 1 | 2 4 2 4 3 |
| 2 | 2 1 3 1 0 |
| 3 | 1 0 0 0 0 |

2. Build a new table of examples where instead of values of attributes are the frequencies of the values in the corresponding column:

2 4 2 4 3
2 4 3 4 3
2 4 2 4 3
1 1 3 4 2
2 4 3 1 2

3. Find the sums of frequencies of values for examples:

| Exampl | Sum |
|---|---|
| 1 | 15 |
| 2 | 16 |
| 3 | 15 |
| 4 | 11 |
| 5 | 12 |

Now we have done most of the work, and we can stop and explain what we did. Counting the frequencies of occurrence of attribute values we found the measure of conformity of examples – the more frequent value -- the more common object it possibly

represents. Hence, the larger the sum of frequencies in the last table, the more common the example. By substituting frequencies for values of attributes we built a system with finite number of elements where each element has a measure. The measures constitute a weakly monotonous system – if a measure of an element increases (decreases) then the measures of other elements change in the same direction or remain unchanged.

**Theorem**. *(Mullat's theorem).* By repeatedly finding an element with the least sum of frequencies and by excluding it from the set of elements (examples in our case) one obtains a *discriminating sequence of examples*. The ordered tail of the discriminating sequence constitutes a *kernel of the monotonous system* that is the most characteristic subset of examples.

**Remark 1**. If there are several examples with the minimal sum of frequencies then it makes sense to exclude the example closest (by Hamming distance) to the previous excluded example.

**Remark 2**. If there are several examples with equal frequencies at the local maximum of the discriminating sequence then only one (the last one) must be included into the kernel of the monotonous system.

Now we can continue the example, knowing what we have to do:

4. Find the example with the least sum of frequencies – number 4 in our case. Exclude the example from the set of examples and repeat the computations for the remaining set of examples.

By repeating the computations we get each time new sums of frequencies that are the following:

| Example | Sum |
|---|---|
| 1 | 14 |
| 2 | 14 |
| 3 | 14 |
| 5 | 10 |

| Example | Sum |
|---|---|
| 1 | 13 |
| 2 | 12 |
| 3 | 12 |

| Example | Sum |
|---|---|
| 1 | 9 |
| 3 | 9 |

| Example | Sum |
|---|---|
| 3 | 5 |

The last sum is computed only for correctness test – it must be equal to the number of attributes.

This procedure gives us the discriminating sequence of attributes and their measures that are sums of their frequencies computed in step 4. The first element in the sequence is attribute number 4, and its measure is 10, the second element is attribute number 5 etc.

| Attribute number | Measure |
|---|---|
| 4 | 11 |
| 5 | 10 |
| 2 | 12 |
| 1 | 9 |
| 3 | 5 |

The ordered tail of measures of the discriminating sequence are 12, 9, 5 in our case. The respective sequence of examples is 2, 1, 3. This gives us the class that we extracted from examples. The remaining set of examples: 4 and 5 constitute another class. These examples together with their values of attributes can be used now as an input for the algorithm ID3 or AQ in order to build a decision tree or a rule-based classifier.

## 3.6.4. Discovering relations and structure

Discovering relations between objects of the world seems to be the most general way of cognition. Let us have a set of objects *D* and a goal to find unary relations on this set. This goal can be achieved by looking at relations as clusters and to apply some clustering algorithm. The same approach can be applied for binary relations, taking the set $D \times D$ as the set of elements where clusters are built. This approach has been described by Peeter Lorents in [26].

We propose here a simple procedure of constructing a binary relation that relies on an operator *related(x,y)* for recognizing relationships between the objects. This operator implements a predicate which is true for related objects *x* and *y*. One can argue that this predicate represents a relation that has to be constructed, and this is correct. However, it is easier to answer the question whether two concrete objects are related than to build a relation as a whole in some other way. Let us have a finite set *objects* that includes all objects we have under consideration. Then constructing a relation can be represented by the algorithm *newRel(objects)* that constructs step by step a binary relation *rel* between the elements of *objects*, using the procedure *addElement((x,y),rel)* for adding a pair *(x,y)* to *rel* each time when the pair satisfies predicate *related(x,y). Algorithm of constructing a relation* is presented in A.3.17.

**A.3.17:**

```
newRel(objects)=
rel={};
for x∈objects do
        for y∈objects do
                if related(x,y) then
                        rel=addElement((x,y),rel)
                fi
        od
```

```
        od;
        return rel
```

It is obvious that constructing a relation depends completely on the quality of the operator *related()*. In some sense the algorithm A.3.17 describes a way of perceiving the world that even humans apply. They use the operator *related()* intuitively for making generalizations about relationships between objects. Researchers apply this algorithm consciously and systematically in order to get new knowledge in the form of relations.

This algorithm can be modified so that several relations will be constructed simultaneously, see the idea of simultaneous detection of relations in [26]. One just has to use a set of relations *rels* instead of a single relation, and for each pair of objects select the matching relations where it should be added to. The procedure *related(x,y)* has to be changed to *related(x,y,rel)*. This gives us the algorithm A.3.18 that tests pairs of of objects from the set *objects* and finds the relations in a set *rels* of given relations where these pairs can be added. The complexity of constructing a relation is hidden in the procedure *related(x,y,rel)* that must somehow understand the meaning of each relation, i.e. the different ways how objects can be related. Elements of the set *rels* are initially empty relations.

**A.3.18:**

```
newRels(objects,rels)=
        for x∈objects do
                for y∈objects do
                        for rel∈ rels do
                                if related(x,y,rel) then
                                        rel=addElement((x,y),rel)
                                fi
                        od
                od
        od;
        return rels
```

It is possible to see that this algorithm is in principle a clustering algorithm for a case when clusters represent relations that can intersect. If the relations can not intersect, then sometimes common clustering algorithms can be adapted for learning sets of relations.

Testing of pairs can be generalized to testing of tuples with three or more elements, see Lorents [26]. The approach of testing all possible relationships between the tuples of objects can be in principle used in constructing ternary relations and even relations with higher arity. However, this would be a very time consuming procedure.

Having some background knowledge about the relation to be constructed, one can improve the algorithm. For example, if one knows that the relation is antisymmetric and transitive, then an ordering can be constructed and used for decreasing the number of tests of pairs, because it is true for any $x$, $y$ and $x<y$ that $x<z$ also for any $z$, $y<z$. If the relation is symmetric and transitive, then the algorithm becomes a clustering algorithm, where the clusters are equivalence classes.

Discovering a structure is even more complicated than discovering relations. It is reasonable to assume that we know more in advance then. Let us know finite sets $D_1$, .., $D_k$ that are sets of values of some attributes. We are going to look for relations between some of these attributes. Let us have a set of pairs of $(x,y)$, where $x$ and $y$ are elements form the sets $D_1$, .., $D_k$ as the input. We assume also that for each element of a pair we know to

which set it belongs, i.e. if the sets intersect, then instead of a pair *(x,y)* we should have *((x,$D_i$),(y,$D_j$))*, where $x \in D_i$ and $y \in D_j$. For our convenience we introduce a function *domain(x)* that gives the set where *x* belongs to. Then we can continue to use the pairs *(x,y)* in the algorithm. The goal is to find binary relations between the sets $D_1$, .., $D_k$ and to construct the relations as the algorithm A.3.19 shows it. The *algorithm of finding relations* uses the following notations:

*rels* – a set of relations constructed gradually by the algorithm;
*pairs* -- a set of pairs given as an input of the algorithm;
*domain(x)*—a set $D_i$ to which the element *x* belongs;
*addElement(e,s)* – adds element *e* to set *s;*
*rel(x,y)* – a notation for a relation between the sets *domain(x)* and *domain(y);*
*newRel(D',D'')* -- creates an empty relation between the sets *D'* and *D''*.

**A.3.19:**

```
findRels(pairs)=
    rels={};
    for (x,y)∈ pairs do
        if not rel(x,y)∈ rels then
                rel(x,y)= newRel(domain(x),domain(y));
                addElement(rel(x,y),rels)
        fi;
        addElement((x,y), rel(x,y))
od;
return rels
```

This algorithm starts from an empty set of relations and adds a relation to the set as soon as it finds a pair that belongs to the relation. If more pairs of the same relation appear, it adds them to the relation as well. The seeming simplicity of the algorithm may be misleading. The complexity of detecting relations (detecting a structure) is hidden here in the construction of pairs that we take as input. A problem solved by this algorithm belongs again to a class of clustering problems. Properly adjusted clustering algorithms canbe applied for solving this problem.

## 3.7. Summary

This chapter contains an overview of the most common learning algorithms together with descriptions of some advanced learning methods like, for instance, learning of theories by inverting resolution. The learning methods presented here can be classified into the following classes: simple parametric learning and learning automata, massively parallel learning and symbolic learning.

Algorithms of the first class were developed in the early days of AI and, since then, have remained almost unchanged. But these simple algorithms are widely used in engineering, in particular, in control of devices. They are used also in other learning systems, in particular, for learning in neural nets.

Research in massively parallel algorithms started also in early days of AI research, but the success in this field was not as good as expected. Development of cheap parallel hardware in recent years has opened completely new opportunities, and much new results have been obtained recently. Genetic algorithms are being used in evolutionary

programming. Neural nets are a promising research area that gives good applications in embedded system, pattern recognition, defense of computer systems etc

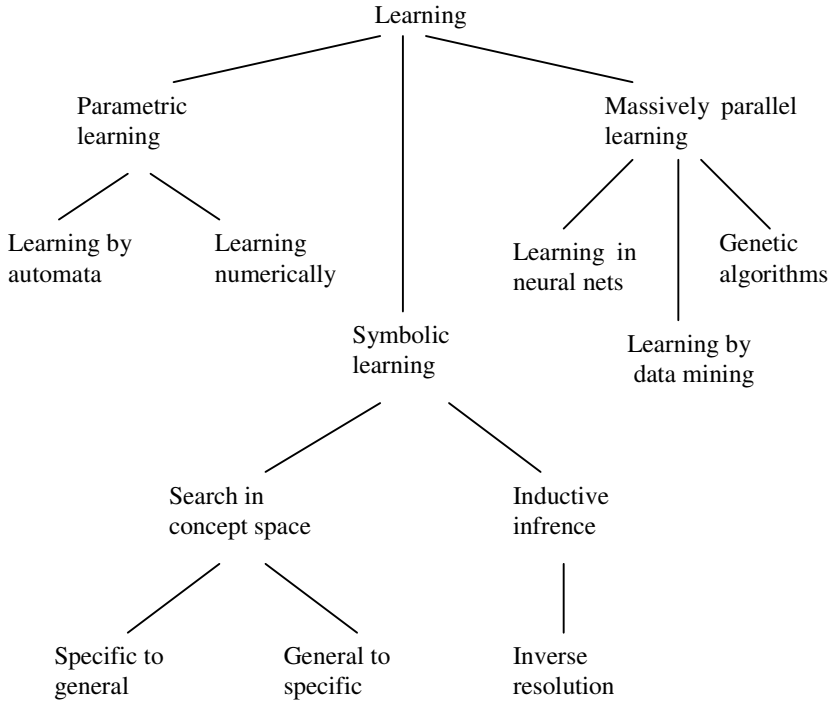



**Figure 3.27**. Hierarchy of learning methods

Symbolic learning is a class of learning algorithms where much work is being done. It gives output to pattern recognition, speech understanding etc. The most recent learning methods are developed in data mining. We have introduced several of them in Section 3.5.3 and 3.5.4. Fig. 3.27 shows a classification tree of the learning algorithms.

## 3.8. Exercises

1. Let us have the following examples: *11001, 10001, 11101.*

   a) Find their least general generalization.

   b) Find a concept that covers the first two examples and does not cover the third one.

2. Describe a perceptron for separating two sets of points on $(x,y)$-plane that are separated by the line $y=2*x$, i.e. the points of the sets that satisfy the inequalities $y>2*x$ and $y<2*x$ respectively.

3. Draw a scheme of two-layer perceptron for four inputs and two outputs.

4. Draw a scheme of a comparator for finding maximum of four analog inputs. Hint: use two comparators for two analog inputs described in this chapter.

5. Calculate the weights for a Hopfield net that separates the binary codes *10100* and *01010*. Draw a scheme of the net.

6. Using ID3 algorithm, build an optimal decision tree for classifying trees from the set of rules given by the following decision table ($x$ means "needles or leaves"):

| | | | | | Large x |
|---|---|---|---|---|---|
| Has needles | Has leaves | | yes | yes | Small x |
| | | | yes | no | Has cones |
| yes | | pine | | | |
| | yes | maple | birch | | |
| no | no | | holyxilon | | |
| yes | | | spruce | | |
| yes | | | | juniper | |

7. Write the inverse substitution of the substitution *((x,a),(y,b))* and term *F(a,g(a),b).*

8. Having background knowledge in the form of the following Horn clauses:

$A \leftarrow P,Q$
$Q \leftarrow R$

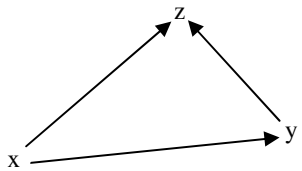
and examples in the form of Horn clauses

$A \leftarrow P,R$

$F \leftarrow R, S$

construct new Horn clauses using identification and absorption operators of inverted resolution.

9. Write the joint probability expression for the following Bayesian network:



# 4. Problem Solving and Planning

Problem solving and planning are the classical AI areas where methods of knowledge handling and search that have been discussed in the previous chapters are being used. Here we introduce, first, methods of problem solving by constraint satisfaction, then we discuss synthesis of algorithms for problem solving and planning of actions. Finally, we briefly introduce architectures of intelligent agents. These parts overlap to some extent, but each of them has its own applications and approaches to the problem solving. Constraint satisfaction and program synthesis are applicable, first of all, to computational problems, planning is a more general approach, and more difficult to represent algorithmically. Agents are complex programs with intelligent behaviour, i.e. with rather universal problem solving capabilities.

## 4.1. Constraint satisfaction problem

A general way of specifying a problem declaratively is to give a set of constraints that must be satisfied by a solution of the problem. This approach is very convenient for a user, who has to specify the problem only, and should not worry about the algorithm of its solution. This approach is becoming more and more popular in many applications like, for instance, in computer aided design, where constraints are often equations and inequalities.

Let us have sets $D_1, D_2, \ldots, D_n$. We call a *relation* a subset of a direct product $D_a \times \ldots \times D_b$, where $a,\ldots,b \in \{1,\ldots,n\}$ are all different. Such a relation is the most general representation of a *constraint*. Each relation $R_i \subset D_a \times \ldots \times D_b$ can be extended to the relation $R'_i$ on the set $D_1 \times D_2 \times \ldots \times D_n$ in such a way that the projection of the extended relation on the set $D_a \times \ldots \times D_b$ will be the relation itself. A *solution of the set of constraints* given in the form of relations $R_1, \ldots, R_k$ is any element of the intersection $S = R_1' \cap R_2' \cap \ldots \cap R_k'$ of extended relations. This is a tuple that contains an element from each set $D_1, D_2, \ldots, D_n$, and elements of it satisfy the relations $R_1, \ldots, R_k$. Another kind of problems on the same set of constraints is finding all solutions of the set of constraints, i.e. finding the intersection $S$ itself. An important variation of the problem is finding a tuple of elements only of some sets $D_p, \ldots, D_q$ which can be extended to a solution defined above.

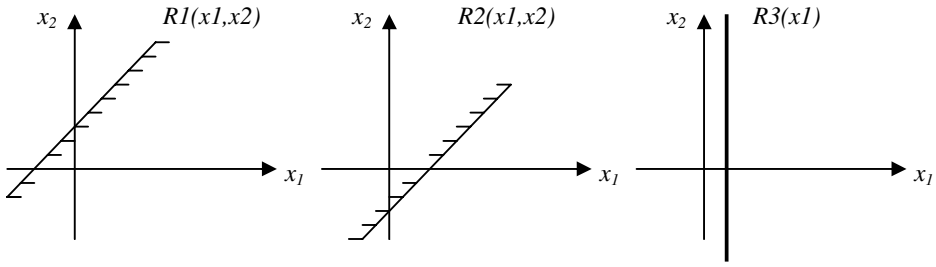



**Figure 4.1.** Three constraints

There are two principally different ways of solving problems on sets of constraints. First, one can transform a set of constraints, simplifying the constraints and preserving solutions, so that at some point it will be easy to find a solution. Second, one can build a

solution gradually step by step, using only one constraint at each step. We will present examples of both approaches to constraint satisfaction problems in this chapter.

We shall use very extensively the structure of the set of relations, i.e. the knowledge telling us which sets actually are bound by one or another constraint. This knowledge can be represented by a graph called a *constraint network*. We also agree that we consider only the problem of finding one tuple as a solution of the set of constraints. We introduce a variable for each set $D_1$, $D_2$, ... ,$D_n$ that takes values from this set, and denote these variables by $x_1$, ... , $x_n$ respectively. Now the problem will be finding values of the variables $x_1$, ... , $x_n$ that satisfy the constraints. Having a relation $R$ as a subset of $D_a \times ... \times D_b$, we shall say that the relation $R$ binds variables $x_a$, ..., $x_b$ that have domains $D_a$,...,$D_b$. Fig.4.1 shows graphs of three constraints $R_1$, $R_2$ and $R_3$ that bind variables $x_1$ and $x_2$. The constraints $R_1$, $R_2$ are linear inequalities and $R_3$ is an equality.

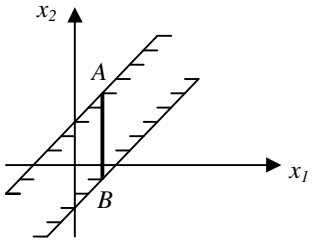



**Figure 4.2.** Solution of three constraints

Any point of the interval $AB$ in Fig. 4.2 can be taken as a solution of the problem specified by the three constarints $R_1$,$R_2$ and $R_3$ shown in Fig. 4.1, because it satisfies all three constraints.

Structure of a set of constraints given as relations can be represented by a bipartite graph, the sets of nodes of which are constituted by variables $x_1$, ... , $x_n$ and relations $R_1$, ... , $R_k$. Edges of the graph correspond to the bindings, i.e. there is an edge between a relation $R$ and a variable $x$ exactly when the relation binds this variable. This is called a *constraint network*.

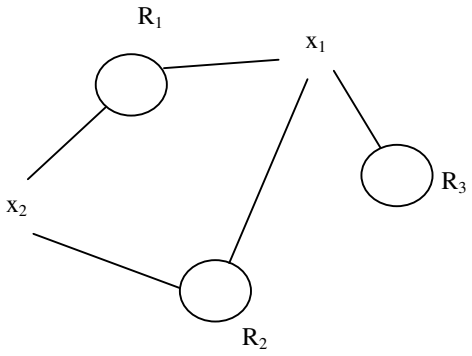



**Figure 4.3.** Constraint network

Fig. 4.3 shows the constraint network of our example where the constraints $R_1$,$R_2$ and $R_3$ bind the variables $x_1$ and $x_2$.

## 4.2. Consistency algorithms

### 4.2.1. Binary consistency

Let $D_1$, $D_2$, ... ,$D_n$ be finite sets. We shall consider here binary relations and denote by $R_{ij}$ a relation which binds the variables $x_i$ and $x_j$ from the sets $D_i$ and $D_j$. This relation can be considered as a mapping from $D_i$ that is called *domain of relation* to $D_j$ that is called its *range of relation*. In the case of constraints as binary relations, the graph representing a constraint network can be simplified. A *binary constraint graph* is a graph with variables as its nodes and constraints as its arcs. It can be obtained from the general constraint network by dropping all constraint nodes and substituting one arc instead of the two edges of each constraint. The direction of the arc is chosen so, that a constraint $R_{ij}$ will be represented by the arc from $x_i$ to $x_j$.

If a tuple of values $(v_1, ... , v_n)$ is a solution of the constraint satisfaction problem, then for any arc $(x_i,x_j)$ of the binary constraint graph the pair of values $(v_i,v_j)$ must satisfy the constraint $R_{ij}$. Therefore, if in the $D_i$ there is an element $v'_i$ which does not satisfy $R_{ij}$ with any element of $D_j$, then the element $v'_i$ can be excluded from consideration when a solution is searched. And, vice versa, if an element $v'_j$ of $D_j$ has no match in $D_i$ that satisfies $R_{ij}$, then the element $v'_j$ can be excluded from consideration. We call this element a *no-match element*. This gives the following consistency condition for arcs. An arc $(x_i,x_j)$ of a binary constraint graph is *consistent*, iff the sets $D_i$ and $D_j$ do not have no-match elements. A binary constraint graph is *arc consistent*, if all its arcs are consistent. The arc consistency is expressed also by the following formula:

$$(\forall u \in D_i \; \exists v \in D_j \; R_{ij}(u,v)) \& (\forall v \in D_j \; \exists u \in D_i \; R_{ij}(u,v)).$$

An arc can be made consistent by removing from $D_i$ and $D_j$ all no-match elements. (Strictly speaking, this changes the relations binding $x_i$ and $x_j$, but it will not influence the solution of the consistency problem.) First, we introduce a procedure *semioin(R,found)* which removes the no-match elements of the domain of the first bound variable of relation $R$ and sets the variable *found* true, if such elements were found. We denote by *dom(R)* and *range(R)* the domains of the first and second bound variable of $R$, i.e. the domain and range of $R$. *inv(R)* is the inverse relation of $R$, i.e. $inv(R_{ij})=R'_{ji}$.

**A.4.1:**

```
semijoin(R,found) =
        found=false;
        for x ∈ dom(R) do
            L: { for y ∈ range(R) do
                    if R(x,y) then exit L
                od;
                dom(R)=dom(R)\{x};
                found=true;
            }
        od
```

The following algorithm makes a binary constraint graph *G* arc-consistent:

**A.4.2:**

```
arcConsistent(G) =
        found=true;
        while found do
                for R ∈ G do
                        semijoin(R,found);
                        semijoin(inv(R),found)
                od
        od
```

One can see that the *semijoin* procedure has to be repeated for constraints several times, because excluding a no-match element from one set may create no-mach elements in other sets. This algorithm for arc consistency can be improved by taking into account that only domains of those constraints must be checked repeatedly, whose ranges have changed. A clever arc consistency algorithm uses an *open* set for keeping track of constraints that have to be checked, a function *var(G)* for initialization of the set *open* with all variables of the constraint graph *G*, *selectVar(G)* for selecting a variable from G, and a function *pred(y)* which gives a set of variables bound with *y* by a constraint. A constraint binding *x* and *y* is denoted here by *R(x,y)*. The new arc consistency algorithm uses again the algorithm *semijoin(R,found)* for removing the no-mach elements.

**A.4.3:**

```
open=var(G);
while not empty( open) do
        y=selectVar(open);
        for x∈ pred(y) do
                semijoin (R(x,y),found);
                if found then open=open∪{x} fi
        od;
        open=open \ {y}
od
```

## 4.2.2. Path consistency

Let us consider a network shown in Fig. 4.4. Even when all its arcs are consistent, there is no guarantee that the elements chosen from $D_i$, $D_j$ and $D_m$ will satisfy together all three constraints $R_{ij}$, $R_{im}$ and $R_{mj}$. These constraints will be satisfied only, if the graph is arc consistent, and for each pair of values *u* and *v*, where $u \in D_i$ and $v \in D_j$, there exists a value $w \in D_m$ such that $R(u,m) \wedge R(m,v)$ is true. This can be written as

$$\forall u \in D_i \ \forall v \in D_j \ (R_{ij}(u,v) \supset \exists w \in D_m \ R_{im}(u,w) \wedge R_{mj}(m,v)).$$

If this condition is satisfied for any path of length 2 in a graph representing a constraint network, the network is called *path consistent*, or *3-consistent*, considering the number of arcs. This means that three consecutive nodes chosen along any path are consistent with respect to the constraints between these nodes on the path.

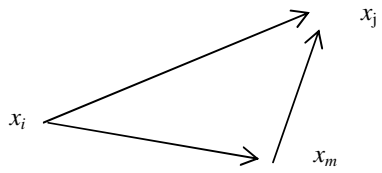



**Figure 4.4.** Path consistency

Path consistency can be generalized for any number of nodes in the following way. A network is *k-consistent*, iff given any instantiation of *k-1* variables satisfying the condition that all the direct constraints among those variables are satisfied, it is possible to find an instantiation of any *k*-th variable such that *k* values taken together satisfy all the constraints among the *k* variables. A network is strongly *k*-consistent, iff it is *j*-consistent for all $j \leq k$. Even if a network with *n* variables is strongly *k*-consistent for $k<n$ there is no guarantee that a solution exists.

## 4.3. Propagation algorithms

### 4.3.1. Functional constraint networks

A relation that binds variables *u,...,v, x,...,y* is called a *functional dependency* with input variables *u,...,v* and output variables *x,...,y*, iff for any tuples $t_1$, $t_2$ of values of bound variables of the relation from the equality of respective values for *u,...,v* in these tuples follows the equality of respective values for *x,...,y*. *Functional constraint network* is a constraint network that contains only functional dependencies as constraints.

Because functional constraint networks are easy to use, it is always reasonable to try to represent a constraint network as a functional constraint network. It can be in principle always done by considering, instead of variables *x,...,y* with domains $D_x,...,D_y$, new variables *x',...,y'* which have powersets of $D_x,...,D_y$ as their domains. However, the new domains are exponentially larger. Still, sometimes this can be practically done for networks with very small finite domains of variables.

Functional constraints appear also as presentations of equations and structural relations. If an equation

$$f(x,...,y,z,..,w)=0$$

is solvable for the variables *x,...,y*, then it can be considered as a collection of functional dependencies -- one for each variable of *x,...,y*. A structural relation which binds a structured variable *x* with its components $x_1,...,x_n$ gives *n+1* functional dependencies: one for computing the value of *x* from values of its components, and one for each component for computing its value from given value of *x*. Fig. 4.5 shows a functional constraint network. Arrows show the possible information flow during computations, i.e. they go from input variables to constraints and from constraints to output variables.

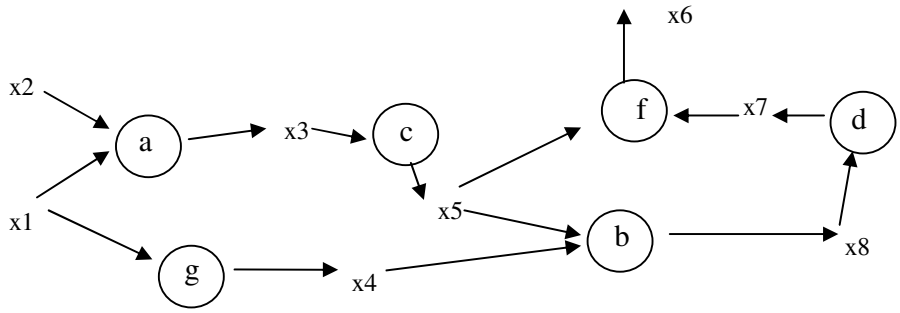



**Figure 4.5.** Functional constraint network

This network can be used for problem solving in the following way. Let us assume that values of some variables are given, let these be values of *x3* and *x4*. Then we can ask to find a value of some other variable, e.g. *x6*. In this case we can say that we have a goal *x3, x4* $\rightarrow$ *x6* that means "compute *x6* from given *x3* and *x4*". Following the arcs on the scheme we find that values of variables *x5, x8, x7, x6* can be computed, hence the goal to compute *x6* can be achieved. Plan for solving the problem will be to apply constraints *c; b; d; f* in the given order.

If we wish to be more precise, then we can write out the algorithm as a sequence of assignments, i.e. a sequential imperative program:

```
x5=c(x3);
x8=b(x4,x5);
x7=d(x7);
x6=f(x5,x7);
```

or as a term

*f(c(x3),d(b(x4),c(x3)))* .

The required value of variable $x_6$ will be computed by the synthesized program. In this way we have solved a problem of the program synthesis on a functional constraint network.

## 4.3.2. Computational problems and value propagation

Let us have a functional constraint network with a set of variables *X*. For any two sets of variables $U \subseteq X$ and $V \subseteq X$ the notation $U \rightarrow V$ will denote a problem "find values of *V* from given values of *U*". We are going to call this a *computational problem* with input *U* and output *V* on the functional constraint network. It must be noted that a computational problem does not necessarily mean numeric calculations. For example, output of a problem may be a response to a query that is given as an input of the problem, or it may be a process on a simulation model etc.

The computability calculus described in the first chapter is immediately applicable for solving computational problems on functional constraint networks. Having a network with variables *x,...,y,* one can build a computability calculus with the alphabet consisting of the symbols *X,...,Y* which denote computability of the variables *x,...,y*. Each functional constraint of the network will be presented by an axiom of the calculus. A problem of the following form: "Given values of the variables *a,...,b* and a functional network *N*, find values of the variables *c,...,d*" is soved by deriving the object

$$\{A,...,B\} \rightarrow \{C,...,D\}$$

in the computability calculus, and by applying its meaning (the function derived) to the given values of *a,...,b*. This algorithm is applicable under the conjecture that every function that is found for computing some variable, computes it correctly, i.e. if there are several ways of computing a variable, then all these ways are acceptable.

The derivation algorithm A.1.2 for computability calculus from the Chapter 1 can be improved in the following way. We associate a counter with each axiom of the calculus, that shows, how many input elements of the axiom still are not computable. When a new element becomes computable, counters of all the axioms that have this element as an input, must be decreased and, if some counter becomes 0, the respective axiom must be applied. If the set of axioms is regarded as a functional constraint network with counters at the constraint nodes, each arc of the network must be processed only once - when the value passed along this arc becomes known. This gives us a possibility to build a *fast constraint propagation algorithm* A.4.4 with linear time complexity with respect to the number of arcs in the constraint network.

The following notations are used in the algorithm for computing values of the variables *out={c,...,d}* from given values of the variables *in={a,...,b}*:

*known* - set of variables already known to be computable;
*open* - set of unprocessed variables from the set *known;*
*count(r)* - counter of the constraint *r* showing how many input variables of the constraint are still unknown;
*countdown(r)* - decreases the value of the *count(r)* by one;
*initcount()* - initializes the values of all counters to the numbers of input variables of constraints;
*succ(e)* - successors of the node *e* in the constraint network;
*plan* - sequence of applied constraints which is a plan for computations;
*takeFrom(s)* - produces an element of the set *s* and excludes it from *s*.

**A.4.4:**

```
known=in;
plan=();
open=in;
initcount();
while not empty(open) do
        if out⊆known then success() fi;
        x=takeFrom(open);
        for r∈succ(x) do
                if count > 1 then
                        countdown(r)
```

```
                elif count(r) == 1 & succ(r) ⊄ known then
                        plan=append(plan, r);
                        open=open∪(succ(r)\known);
                        known=known∪succ(r);
                        countdown(r)
                fi
        od
od;
failure()
```

## 4.3.3. Equational problem-solver

The propagation algorithm A.4.4 can be adopted also for constraints represented as equations. In the simplest case, we shall assume that each equation can be solved with respect to any variable in it. The successor function *succ(r)* for an equation *r* gives a set of all its neighbors. This gives us the following *equational problem solving* algorithm.

**A.4.5:**

```
plan=();
known=in;
open=in;
initcount();
while not empty(open) do
        if out⊆ known then success() fi;
        x=takeFrom(open);
        for succ(x) do
                if count(r) > 1 then
                        countdown(r)
                elif count(r) == 1 & succ(r) ⊄ known then
                        plan=append(plan, r);
                        open=open ∪ (succ(r)\known);
                        known=known ∪ succ(r);
                        countdown(r)
                fi
        od
od;
failure()
```

This algorithm can be generalized for the case where only some of the variables bound by an equation can be computed from it. Now we shall need two sets of variables for describing an equation:

*input(r)* - input variables of *r*,
*bound(r)* - all bound variables of *r*.

We present the algorithm in two parts, preserving the possibility of further improvement of its internal part.

**A.4.6:**

```
plan=();
known=in;
open=in;
initcount();
while not empty(open) do
        if out⊆known then success() fi;
        x=takeFrom(open);
        for r∈ succ(x) do
                DERIVE(r,plan,open,known);
        od
od;
failure()
```

The procedure *DERIVE(r,plan,open,known)* builds plan step by step, adding at each step a pair *(r, succ(r)\known)* that shows not only the constraint *r* but also the variables valuated by this constraint.

**A.4.7:**

```
DERIVE(r,plan,open,known)=
    if count(r) >1 then
            countdown(r)
    elif count(r) = 1 & input(r)⊆known & succ(r) ⊄ known then
            plan=append(plan, (r, succ(r)\ known));
            open=open ∪ (succ(r) \ known);
            known=known ∪ succ(r);
            countdown(r);
    fi
```

The algorithm can be further improved by extending it for structural relations. A structural relation *r* will be represented by a set of its bound variables - *bound(r)* and the structural variable *str(r)* the value of which consists of the values of its component variables. Now we have to show among the parameters of the *DERIVE( )*also the most recently computed variable *x*. We get the new planning algorithm by substituting the new *DERIVE1(r, x, plan, open, known)* instead of *DERIVE (r,plan,open,known)* in A.4.6.

**A.4.8:**

```
DERIVE1(r, x, plan, open, known )=
    if count(r) > 1 & str(r)∉ known then
            countdown(r)
    elif count(r) =1 & succ(r) ⊈ known&( input(r) ⊆ known or x = str(r)) then
            plan=append(plan,( r, succ(r)\ known));
            open=open ∪ (succ(r) \ known);
            known=known ∪ succ(r);
            countdown(r)
    fi
```

This algorithm can solve problems on functional constraint networks with constraints as generalized equations and structural relations. Its time-complexity remains still linear with respect to the number of arcs in a network.

## 4.3.4. Minimizing an algorithm

All value propagation methods described in the previous section either give an algorithm for solving a given computational problem on a constraint network, or tell us that the problem is unsolvable. However, we have not paid attention to the fact that an algorithm developed by means of a value propagation technique may contain excessive steps that are not needed for solving a problem. Indeed, if a problem $U \rightarrow V$ is solvable on a given constraint network, then the value propagation gives us a sequence of steps

$a_1, a_2, \ldots, a_n$

such that after performing the last step, values of all variables from $V$ will be computed. Each step $a_i$ is application of a functional constraint, solving of an equation or using a structural relation. In all these cases we can consider a step $a_i$ as application of a function. Let us denote by $x_i$ the variables used as inputs at the $i$-th step and by $y_i$ the variables that did not have values before the $i$-th step and got the values at this step. The value propagation algorithms add steps to a constructed algorithm, first, on the basis of possibility of application of a function, and second, taking into account that the function computes something new. However, the value propagation algorithms are unable to decide whether the computed values of new variables will be needed for solving the problem.

Having already the complete algorithm, one can decide at each step whether it is needed or not. For this purpose, one has to introduce at each step $a_i$ a set $w_i$ of variables that still ave to be found for concluding the computations. At the last step this set is empty – all need values are known. One step back, in general, at a step $a_{i-1}$, the set of variables needed for concluding the computations is the following:

$w_{i-1} = w_i \cup x_i \setminus y_i.$

We see that the set $w_{i-1}$ is computed at each step $i$ from the information of the $i$-th step – from the set $w_i$ taking into account input and output of the $i$-th step. If at some step $i$ we detect that the function $a_i$ does not compute any new variable, i.e. $y_i \cap w_i$ is empty, then this step can be discarded, this done by the procedure *discard*($a_i$). This gives us the following *program minimization algorithm* in notations used above:

**A.4.9:**

```
w={};
for i=n-1 to 1 do
        if empty(y_i ∩ w) then
                discard(a_i)
        else
                w= w_i ∪ x_i \ y_i.
        fi
od
```

## 4.3.5. Lattice of problems on functional constraint network

At this point it will be interesting to look at the set of all computational problems $U \rightarrow V$ on a given functional constraint network (FCN).

**Definition**. A problem $P1{:}U1 \rightarrow V1$ is greater than a problem $P2{:}U2 \rightarrow V2$, iff $U1 \subseteq U2$ and $V2 \subseteq V1$ and at least one of the inclusions is strict (i.e. $\subset$). This denoted by $P2 < P1$.

This definition is justified by the fact that it is obviously not more difficult to solve a computational problem, when more inputs are given ($U1 \subseteq U2$). And from the other side, it is easier to compute less output values ($V2 \subseteq V1$). Hence, the problem $P1$ is greater than $P2$. Now we have defined a *partial order of problems.*

**Definition**. *Union of problems* $P1{:}U1 \rightarrow V1$ and $P2{:}U2 \rightarrow V2$ denoted as $P1 \cup P2$ is the problem $U1 \cap U2 \rightarrow V1 \cup V2$.

**Definition**. *Intersection of problems* $P1{:}U1 \rightarrow V1$ and $P2{:}U2 \rightarrow V2$ denoted as $P1 \cap P2$ is the problem $U1 \cup U2 \rightarrow V1 \cap V2$.

The problems on a functional constraint network constitute a *lattice of problems*. This is a set with two operations $\cup$ and $\cap$ where for any elements $a$, $b$ of the set also the elements $a \cup b$ and $a \cap b$ belong to the set. The lattice includes always the smallest problem problem $X \rightarrow \varnothing$ and the largest problem $\varnothing \rightarrow X$, where $X$ is the set of all variables of the FNC.

**Theorem**. For any two problems $P1$ and $P2$ where $P1 < P2$, if $P2$ is solvable on a FCN, then the problem $P1$ is also solvable on the same FCN.

Proof: any algorithm of solving $P2$ solves $P1$ as well, because $V1 \subseteq V2$.

Let us look now at the number of all possible problems on FCN. This is the number of elements of the lattice of problems. This number is very large. Indeed, let us have a set of variables of FCN that has $n$ elements. One can take any of its $2^n$ subsets as a set of inputs of a problem and any of its subsets as a set of outputs of a problem. Hence the total number of problems on FCN with $n$ variables is $2^{2n}$.

We are mostly interested in solvable problems. This set may be much smaller. This set includes *maximal solvable problems*. These are the most interesting problems for us, because knowing how to solve them one can solve any solvable problem on the FCN. The number of maximal problems is obviously less than $2^n$, because this is the number of possible input sets of problems.

## 4.3.6. Higher-order constraint propagation

Any functional constraint can be represented by a program that computes its outputs from inputs. But programs can represent even more complicated constraints - *higher-order functional constraints*.

Let us consider two programs that represent the constraints $a$ and $b$ shown in Fig. 4.6. The first program has input variables $x$, $y$ and an output variable $z$. The second program has an input variable $s$, an output variable $t$. But it has one more input – a procedural parameter $g$ that takes functions as values. Let us assume that this parameter can take

values which have input *u* and output *v*. Fig. 4.6b shows this input as a constraint between the variables *u, v* respectively. Arrows in Fig. 4.6b show possible directions of dataflow during the application of the constraint *a*.

b

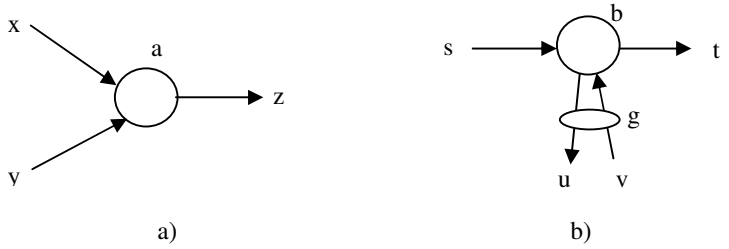



**Figure 4.6.** Programs as constraints

Observing the dataflow of the second program, we distinguish its input and output data (i.e. the values of the variables *s* and *t)* and the data passed between this program and some program *f* that must be given as a value of its parameter *g*. The program *b* produces the input of *f* (a value of the variable *u*) and gets back a value of the variable *v* that is output of *f*. This happens every time when *f* is called, and that, in its turn, depends on the computations performed by the program *b*. We must distinguish this dataflow, which occurs during the subcomputations performed by the program *b*, from the "ordinary" input and output of programs. This is reflected in markings of arcs binding the variables *u, v* that differ from the arcs binding other variables with programs. The role of the higher order variable *g* is binding the variables *u* and *v* as input and output of the same subcomputation. It will not be represented as a node of constraint network. In general, a program can have more than one procedural parameter, as well as several input and output variables.

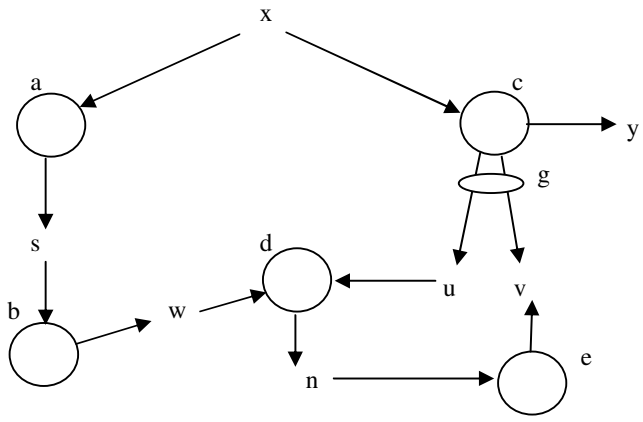



**Figure 4.7.** A higher order constraint network

A *higher order constraint network* is a graph whose nodes are variables and constraints. The constraints are functional or higher order dependencies represented by programs (or even by equations etc.). Its arcs bind variables with constraints as it has been shown in Fig.4.6. It is remarkable, that this graph is still a bipartite graph with the nodes divided into disjoint sets of variables and constraints (procedural parameters are not nodes of this graph). An example of a higher order constraint network is in Fig.4.7. Such a network can be used as a program specification, provided that a goal is given in the form of lists of input and output variables of the desired program. For instance, Fig. 4.7 becomes a program specification as soon as the realizations of its constraints are given and we say that $x$ is the input and $y$ is the output of the program desired.

It is quite easy to build a program for computing $y$ from $x$ on the constraint network shown in Fig. 4.7. Two possible schemas of such a program are shown in Fig.4.8 a, b where we can see that the program implementing the constraint $c$ has a subcomputation which is an application of the constraints $a,b,d$ and $e$ in one case and of the constraints $d$ and $e$ in another case.

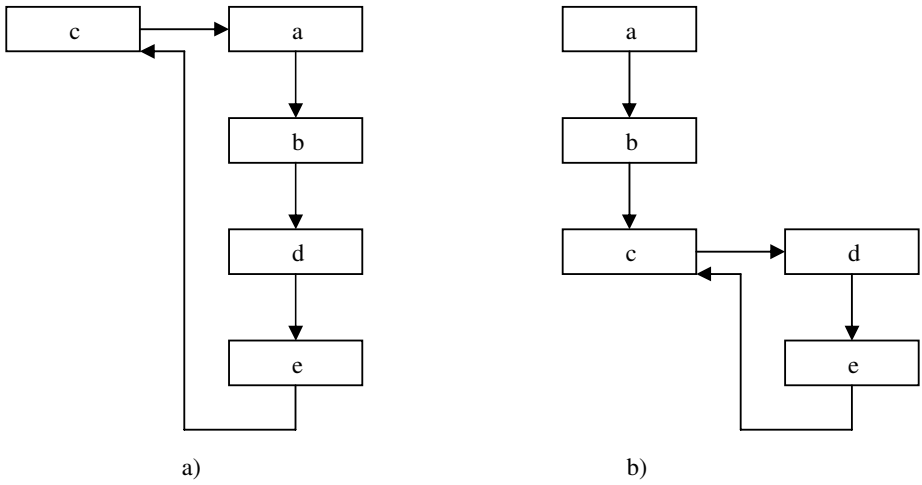



**Figure 4.8.** Programs on constraint networks

We can extend the language of the computability calculus introduced earlier in Chapter 1, so that it enables us to represent also higher order constraints. To do this, we shall allow the sets on the left side of arrow to contain objects of the form $U \rightarrow V$ where $U$ and $V$ represent the inputs and outputs of a procedural parameter. The object $U \rightarrow V$ on the left side of the main arrow is called a *subtask*. Let us have a higher-order constraint with the sets of inputs $\{x_1,..,x_m\}$ and outputs $\{y_1,...,y_n\}$ which also has a subtask (a procedural parameter). Let this parameter in its turn have inputs $u_1,...,u_s$ and outputs $v_1,...,v_t$. Then the symbolic representation of this constraint will be

$$(U_1,...,U_s \rightarrow V_1,..,V_t),X_1,...,X_m \rightarrow Y_1,...,Y_n$$

(we have dropped the curly brackets denoting the sets in this formula, because these sets are already distinguished syntactically by arrows and parentheses.)

If we wish to use a computability calculus for planning of computations on constraint networks with higher order functional constraints, we have to extend the computability calculus presented in Chapter 1 so that it will be applicable also for higher order functional constraints. For this purpose we introduce a new derivation rule *(3)* for applying subtasks, and get the following set of rules:

$$\frac{A \to B \quad B \cup F \to D}{A \cup F \to B \cup D} \qquad (1)$$

$$\frac{A \to B \cup F}{A \to B} \qquad (2)$$

$$\frac{(A \to B) \cup C \to D \qquad A \cup F \to B}{C \cup F \to D} \qquad (3)$$

where A, B, D denote sets of variables and C denotes a set of variables and subtasks. Practical usage of this calculus appears inefficient, if one does not use some additional heuristics. Therefore a special algorithm for planning on these networks has been developed that takes into account the specifics of the networks.

Let us consider first the evaluation of variables on a constraint network that contains only one higher-order functional constraint with a single subtask, like the network in Fig. 4.7, for instance. Then the following steps are performed.

First the procedure of simple value propagation is done using only functional constraints that are not higher-order. If this does not solve the problem (does not give values of all outputs of the problem), then a higher-order functional constraint (*hofc*) is applied, if it is applicable. A hofc is applicable if and only if all its inputs are given and all its subtasks are solvable and it computes values of some variables that have not been evaluated yet. A sequence of applicable functional constraints obtained in this way is called *maximal linear branch (mlb).* It contains one hofc at the end of the sequence. There are two possible outcomes of this procedure:

- A mlb cannot be found and the problem is unsolvable.
- The constructed mlb reduces the problem to a simpler one. (If a mlb exists, then it obviously reduces the problem, because it computes some new values of attributes.)

Now the simple value propagation (using only functional dependencies that are not higher-order) is done again. The problem is either solved in this way, or it is unsolvable, because no more possibilities for computing new values exist. It is easy to see that the algorithm above works in linear time.

If there is more than one higher-order dependency, then there are several options for choosing a hofc, and the planning algorithm cannot choose any appropriate hofc for solving the computational problem without deeper analysis. Then the exhaustive search on and-or tree is required.

There are two basic strategies of construction of and-or trees for solving subtasks. The first one does not allow to use hofcs that have already been used in some mlb. Having this restriction we reduce the number of branches in the search tree to the number of all

possible permutations of hofcs. The second strategy permits repetitions and then the algorithm is PSPACE-complete, see the remark at the end of this section and also [47].

In a general case, when constraint network contains several higher-order functional constraints, the evaluation algorithm is as follows. First the procedure of simple value propagation is performed considering all constraints that are not higher-order. If this does not solve the problem (does not give values of all outputs of the problem), then any applicable hofc is chosen and a maximal linear branch is obtained. There are three possible outcomes of this procedure:

- After constructing the mlb the problem is solvable.
- A mlb cannot be found and the problem is unsolvable.
- A mlb can be found and the initial problem $U1 \rightarrow V1$ is reduced to a simpler one $U2 \rightarrow V2$, where $U2 = U1 \cup Y$ and $V2 = V1 \backslash Y$, where Y is the set of outputs of the hofd.

This procedure (construction of mlb) is repeatedly applied until the problem is solved or no more mlb can be constructed. It is important to notice that for applying a hofc we have to solve all its subtasks. This means that the whole procedure of problem solving must be applied for every subtask. As stated above, this requires a search on an *and-or tree of subtasks* on the constraint network. The root of a tree corresponds to the initial problem, and it is an or-node, because there may be several suitable mlbs for this problem. And-nodes correspond to higher-order functional constraints and have one successor for its each subtask, plus one successor for the reduced task that has to be solved after applying the mlb. Or-nodes of the tree correspond to the subtasks that have to be solved for their parent and-node. This is a good example of applying a general search algorithm (search on and-or trees) for solving a specific search problem (higher-order constraint propagation) that determines the search tree structure.

The algorithm A.4.10 of search on and-or tree of subtasks is presented below. This algorithm is presented by two functions. The first function *hoSolver* operates in or-nodes and selects a branch, i.e. the hofc that can be used at the current or-node for solving the problem. First, this function tries to solve the problem by calling *linPlan* that is the algorithm for solving problems without higher order functional constraints. If the problem is not solvable by *linPlan,* it calls the other function *expand* for each subtask of a selected hofc. The function *expand* expands a given partial plan with a new part that includes an algorithm for solving each subtask of the selected hofc, and in addition, an algorithm for solving the problem that remains after application of the selected hofc. The following notations are used in the algorithm:

*cn* – constraint network

*plan* – the algorithm that is a list of lists of applicable constraints constructed stepwise (one list for each subtask)

*solves(plan,S)* – a predicate that is true iff *plan* solves the problem *S*

*hoConstraints (cn)* — all hofc of the constraint network *cn*

*subplan* – a list including an algorithm for solving each subtask of the selected hofc, and icluding an algorithm for solving the problem that remains after application of the selected hofc

*subtasks(c)* – all subtasks of hofc *c*

*adjust(S,plan)* – a procedure that changes the given task *S*, taking into account all values already computed by *plan*.

**A.4.10:**

```
hoSolver(S,cn)=
        plan=linPlan(S,cn);
        if solves(plan,S) then success() fi;
        for c ∈ hoConstraints (cn) do
                expand(plan,c,S);
                if solves(plan,S)then success()fi
         od;
         failure()


expand(plan,c,S)=
        subplan=();
         for S1∈ subtasks(c) do
                p= hoSolver(adjust(S1,plan),cn));
                if solves(p,S1 )then
                        p= (subplan,(p))
                else
                        failure()
                fi
        od;
        p= hoSolver(adjust(S,plan),cn));
        if solves(p,S )then
                p=(plan,(p))
        else
                 failure()
        fi;
```

If one substitutes the function *expand* instead of its call in *hoSolver,* then one can see the similarity of the algorithm A.4.10 presented here and the algorithm A.2.18 for search on and-or trees The present algorithm is adjusted for the search tree that appears during the search on a constraint network. Therefore it includes more details.

## 4.4. Special algorithms of constraint solving

### 4.4.1. Clustering of equations

The equational problem solver described above does not solve systems of equations, if the equations must be solved simultaneously. For example, even the system of two rather simple equations

$x+y=2$
$x*y=1$

can not be solved by the algorithms presented above.

From the other side, application of standard solution methods of systems of equations to constraint networks that are large systems of equations is inefficient. Therefore

we shall describe a method of *clustering of equations* – transforming a system of equations into a network of clusters of constraints each of which is a minimal system of equations that has to be solved simultaneously (Karzanov, Faradjev [20]). The whole system of equations can be solved then by solving the clusters one by one sequentially.

Let us consider a constraint network that is a system of equations with the structure shown in Fg. 4.9. Constraint propagation is impossible in it, because there is no equation with a single unknown variable to start with. However, if the equations are good, e.g. they are independent linear equations (which cannot be decided from the schema alone), then the system can be solved and values of variables that satisfy all the constraints can be found.

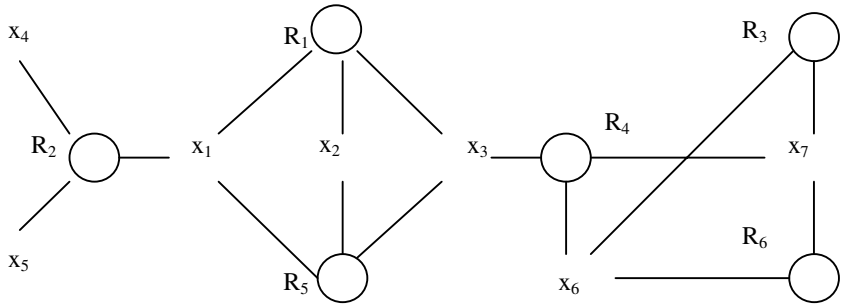



**Figure 4.9.** System of equations for simultaneous solving

A sufficient structural condition for the solvability of a system of equations is that one can establish a bijective (one-to-one) mapping between variables and equations. The algorithm of building maximal mapping between the variables and equations has the time complexity $O(n^2)$ with respect to number of nodes *n* of constraint network, hence it is easy to do. This is done by constructing paths of maximal length that does not contain multiple instances of any node as the following *algorithm of building a mapping between variables and equations* shows it.

**A.4.11:**

1. Take any edge *(x,C)* with unmapped *x* and *C* as an initial path, and define the mapping to consist of the single pair *(x,C)*.

2. Extend the path by adding an edge *(C,x')* where *x'* does not belong to the path. Thereafter extend the path by adding an edge *(x',C')* where *C'* doesn't belong to the path and add the pair *(x',C')* to the mapping. Add the edges in this way as long as possible.

3. Extend the path from the other end analogously and add every second edge to the mapping.

4. If at least one of the edges at the end of the path already belongs to the mapping, then this is a maximal mapping. Otherwise, change the mapping and take into it just the edges that didn't belong to it before.

5. Repeat the steps 1 - 4 until there are no more edges (x,C) with both x and C unmapped. The constructed mapping is a maximal mapping.

Fig.4.10 shows with thick arrows a maximal mapping on the network of our example, as well as orientations on the edges that give us a directed graph. If a satisfactory mapping is found, then the next question is, how to divide the whole system into smaller

parts each of which is a simultaneously solvable system of equations in such a way that the system as a whole is solvable by constraint propagation.

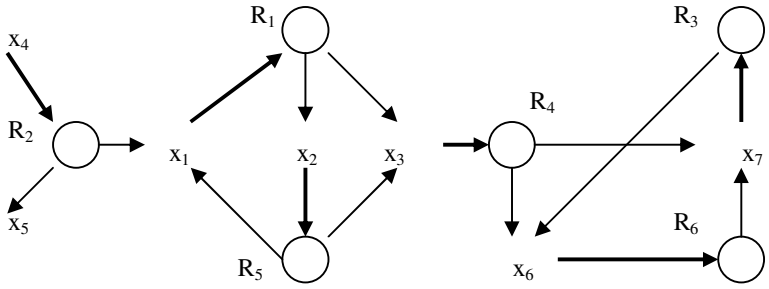



**Figure 4.10.** Maximal mapping

Now we can present the *algorithm of clustering of equations* as follows:

**A.4.12:**

1. Build a maximal mapping of equations and variables.

2. Orientate the edges of the network so that each edge that binds a variable $x$ with a constraint $C$ according to the established mapping becomes an arc $(x,C)$ and all other edges become the arcs of the form $(C',x')$, where $C'$ is a constraint and $x'$ is a variable.

2. Find strongly connected parts $P1,...,Pk$ of the directed graph obtained (these are the parts where every pair of nodes is connected by a path). Each such part represents a minimal system of equations that must be solved simultaneously.

3. Order the parts $P1,..., Pk$ in the ordering opposite to the directions of arcs which bind them. This gives the correct order for constraint propagation between $P1,...,Pk$.

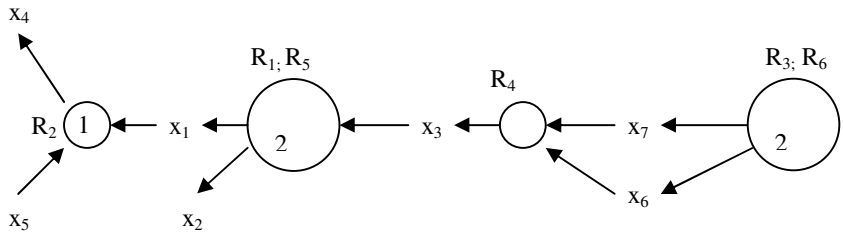



**Figure 4.11.** Resulting constraint network

Fig.4.11 shows the resulting constraint network where the constraints themselves are systems of equations. The numbers in circles of constraints are their ranks, i.e. the numbers of variables computable from each constraint. It can be also seen from the figure that the variables $x_4$ and $x_5$ can not be computed, or $x_4$ can be computed only if a value of $x_5$ will be given. This is because the variable $x_5$ does not belong to the maximal mapping constructed during the clustering of equations as shown in Fig.4.10.

Now we can generalize a little and say that a problem of finding values of variables *x1,...,xm* from a system of equations is (structurally) solvable iff variables accessible from *x1,...,xm* on the oriented graph built according to the first step described above are all mapped by one-to-one mapping built between variables and equations.

## 4.4.2. Interval propagation

An interval can be considered as a value of an interval variable *x* that is a pair:

*lx* - lower bound of values of *x*
*ux* - upper bound of values of *x*.

Problems on constraint networks with interval variables can be solved by *interval propagation*, if the constraints are monotonous functional dependencies and linear equations. The latter can be splitted into the functional dependencies of the following form:

$a = b + c$
$a = b - c$
$a = b * c$
$a = b / c$.

Each of these dependencies can be treated algorithmically for interval variables by using the following dependencies, where *l* in an identifier denotes lower and *u* denotes upper bound of an interval

If $a = b + c$ then *la=lb+lc, ua=ub+uc*
If $a = b - c$ then *la=lb-uc, ua=ub-lc*
If $a = b * c$ and $a>0$ and $b>0$ then *la=lb*lc, ua=ub*uc*
If $a=b/c$ *c* and $a>0$ and $b>0$ then la=lb/uc, ua=ub/lc.

For negative values in multiplication and division one must take into account also the signs of the bounds, e.g. if $a = b*c$, $uc<0$ and $lb>0$ then *la=lc*ub, ua=uc*lb,* because *la* and *ua* both are negative. A difficulty arises only with the dependency for division, if an interval value for *c* includes *0.* Then we cannot decide about the bounds at all.

Fig. 4.12 shows *classification of constraint solving problems*. Two large classes are constituted by constraints on finite domains and infinite domains. For the constraints on finite domains the search is done mainly on the sets of values of variables. In the case of infinite domains, the search takes more into account a structure of the constraint network. The figure includes even several classes of problems that we did not discuss here: problems on subdefinite models, handling inequalities, rational trees [9] and constraint hierarchies [10]. These problems are well discussed in literature.

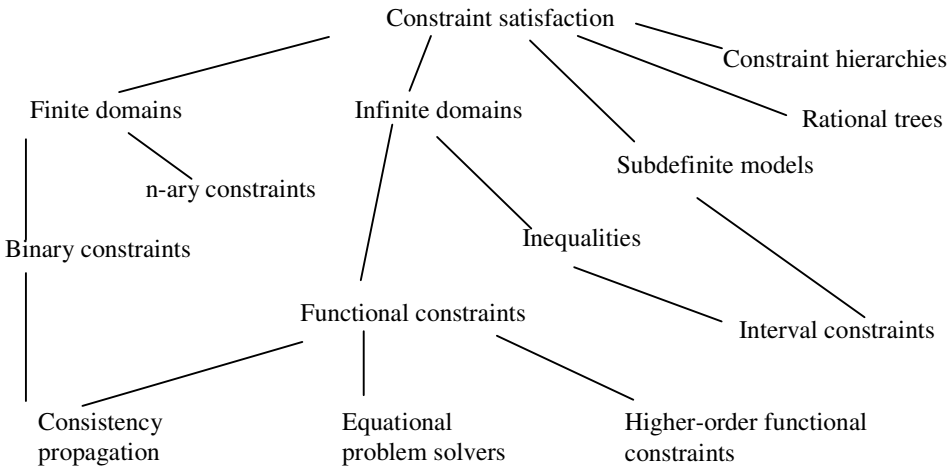



**Figure 4.12.** Classes of constraint solving problems

## 4.5. Program synthesis

Program synthesis as an AI research area emerged quite early – in the sixties of the last century [15]. During the first ten years, results were obtained in three directions of program synthesis: deductive, inductive and transformational synthesis. These are main directions of this field even today. A *program synthesis problem* is constructing a program from a given specification of a problem to be solved, which together with background knowledge must include all knowledge needed for solving the problem. The obvious advantage of using this technique in software development is the guarantee that one gets a correct program from a correct specification of a problem.

### 4.5.1. Deductive synthesis of programs

A specification for *deductive synthesis of programs* is represented as a statement in a logic that the problem is solvable (i.e. that a program with required properties exists). Additional knowledge about problems and their solution methods is also presented in logical form. This is done by giving the needed general knowledge in logic and also associating a specification with each program *P* that may be used as a part of the resulting program (of the program that has to be synthesized) as shown in Fig. 4.13.

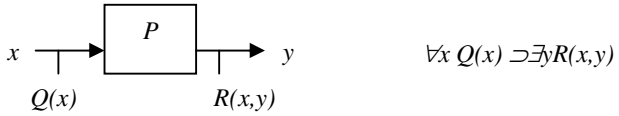



**Figure 4.13.** Program and its specification

The program *P* takes an input *x* and it computes an output *y*. It has a specification in the following general form:

$\forall x\ Q(x) \supset \exists y R(x,y)$

We can distinguish a *precondition Q(x)* of the program in the specification. The precondition tells us which input *x* is acceptable, i.e. the predicate *Q(x)* tells what properties *x* must have. The specification tells us also which properties the output *y* will have, taking into account the input *x*. This is given by the *postcondition R(x,y)* of the program. For the resulting program we know only its specification. Deriving its specification is the logical part of the deductive program synthesis. Deductive synthesis of programs includes two steps:

- Prove the theorem: "a solution of the given problem exists" , i.e. derive the specification of the resulting program.
- Extract a program from the proof.

If the proof is constructive, then by definition, it includes information about constructing the solution, and extracting a program from the proof should be only a technical problem [28]. The difficulty lies in the complexity of proof-search in the given theory. Conventional provers of logic are inefficient for constructing long proofs needed for constructing programs of real size.

## 4.5.2. Inductive synthesis of programs

In the case of *inductive synthesis of programs* a specification is a collection of examples of input-output pairs or even of example computations [4]. Examples of specifications for inductive synthesis are:

1.*(A B C D)* gives *(D C B A),* construct a Lisp program for lists.

1 gives 1
2 gives 4
3 gives 9
…
construct an arithmetic program.

The method of inductive synthesis is inductive inference that we have described in Chapter 3. In particular, inductive logic programming that we have considered in Chapter 3 belongs to inductive program synthesis.

## 4.5.3. Transformational synthesis of programs

Specification for *transformational synthesis of program* is closer to program than in other cases. The synthesis is step-by-step transformation of the specification so that finally it becomes a program [7]. The required knowledge is represented in the form of transformation rules (special form of production rules described in Chapter 1). This form of synthesis is often performed interactively, in cooperation of user and computer.

One way of transformational program synthesis is briefly the following. A specification is given as a set of equations. The equations are unfolded into more explicit

representation of objects and steps of computations. The unfolded entities are rearranged and transformed, and thereafter folded into a program.

## 4.6. Structural synthesis of programs

*Structural synthesis of programs* (SSP) is a special case of the deductive program synthesis. In this section we present it in more detail as an example of practically useful program synthesis method that has been implemented in several software tools.

Having an ontology that includes variables of a problem domain where the programs are synthesized, we can do the following.

- First, we assume that the pieces of programs available for constructing the resulting program are correct.
- Second, we assume also that we know which piece of program can be used for computing which domain variable. (This is how web services are specified today, for instance, by describing their inputs and outputs by means of names of entities from ontology.)

In this case we can use pre- and postconditions of programs in the following simplified form where the postcondition $R(y)$ does not bind the output value with input:

$$\forall x\ Q(x) \supset \exists y R(y)$$

The precondition $Q(x)$ tells only that a proper value of $x$ exists and the postcondition $R(y)$ tells only that the proper value of $y$ will be computed, if the program $P$ is applied. In this case we use only structural properties of programs, and ignore the actual properties of the functions that they implement. This is why the method is called structural synthesis of programs.

The specifications of programs used in SSP are more complex that presented above. First, all input and output variables of programs must be specified by respective pre- and postconditions. This gives us specifications of the form

$$\forall x_1\ x_2\ \ldots\ x_m\ Q_1(x_1) \wedge Q_2(x_2) \wedge \ldots \wedge Q_m(x) \supset \exists y_1\ y_{2\ldots}\ y_n R_1(y_1) \wedge R_2(y_2) \wedge \ldots \wedge R_n(y_n)$$

where the program has $m$ inputs and $n$ outputs. Second, we accept also functional inputs. For instance, a functional variable $\varphi$ can be an input of the program. This gives the following specification of the program:

$$\forall \varphi\ x_1\ x_2\ \ldots\ x_m\ F(\varphi) \wedge Q_1(x_1) \wedge Q_2(x_2) \wedge \ldots \wedge Q_m(x_m) \supset \exists y_1\ y_{2\ldots}\ y_n R_1(y_1) \wedge R_2(y_2) \wedge \ldots \wedge R_n(y_n)$$

Naturally, there can be more than one functional input variable of a program. This makes a specification longer, but not more complex in principle. Luckily enough, the unary predicates $Q_1(x_1)$ etc. can be considered as propositions saying "proper value of $x_1$ exists" etc. We can denote these propositions simply by $X_1,\ldots,X_m,Y_1,\ldots,Y_n$. This gives a much simper formula now:

$$\forall \varphi\ F(\varphi) \wedge X_1 \wedge \ldots \wedge X_m \supset Y_1 \wedge \ldots \wedge Y_n$$

One can get rid of the functional variable $\varphi$, if one writes its specification instead of this variable. Let the specification be $U_1 \wedge ... \wedge U_s \rightarrow V_1 \wedge ... \wedge V_t$. Then the complete specification becomes

$$(U_1 \wedge ... \wedge U_s \supset V_1 \wedge ... \wedge V_t) \wedge X_1 \wedge ... \wedge X_m \supset Y_1 \wedge ... \wedge Y_n$$

In order to get a constructive proof, it is reasonable to use *intuitionistic logic* for specifications. Looking at the formula above we see that we need only propositional logic, and we need only logic with connectives $\wedge$, $\supset$. This logic is *implicative fragment of intuitionistic propositional calculus (IPC)*. The *inference rules of IPC* are rules for introduction and elimination of connectives, in our case only of $\wedge$, $\supset$. We present the rules in Fig. 4.14 using the metavariables A, B, $A_i$, $i=1,...,k$ for propositional formulas. The rule ($\supset$+) means that if B is derived from A then the formula A $\supset$ B can be derived by this rule.

$$\frac{A_1 \quad \ldots \quad A_k}{A_1 \wedge \ldots \wedge A_k} (\wedge+) \qquad \frac{A_1 \wedge \ldots \wedge A_k}{A_i} (\wedge-)$$

$$\frac{\begin{array}{c} A \\ \vdots \\ B \end{array}}{A \supset B} (\supset+) \qquad \frac{A \quad A \supset B}{B} (\supset-)$$

**Figure 4.14.** Inference rules of IPC for conjunction and implication

Besides that, one needs, in principle, so called *structural rules* for reordering propositional variables and for introducing and eliminating extra copies of them in conjunctions. We do not present these simple rules here. Using the rules above we can derive a goal of the form

$$X_1 \wedge ... \wedge X_m \supset Y_1 \wedge ... \wedge Y_n$$

that is originally the following specification of a program to be synthesized:

$$\forall x_1\, x_2 \ldots x_m\, Q_1(x_1) \wedge Q_2(x_2) \wedge \ldots \wedge Q_m(x) \supset \exists y_1\, y_2 \ldots y_n R_1(y_1) \wedge R_2(y_2) \wedge \ldots \wedge R_n(y).$$

Intuitionistic logic has the good property that interpretation of its formulas gives a precise computational meaning to formulas. In particular, a propositional variable has the meaning as a value. If the propositional variable is given or derived, then its meaning is known or can be computed by a known formula. A conjuction $A \wedge B$ has the meaning as a pair of values $(a,b)$, where $a$ and $b$ are the meanings of $A$ and $B$ respectively. An implication $A \supset B$ has a function for computing a meaning of $B$ from the given meaning of $A$ as its meaning. Meanings of derived formulas are constructed according to quite obvious rules shown in Fig. 4.15. These meanings are shown after colon for each formula in the

rules. This allows one to extract a program from a proof of a goal. This concludes the explanation of SSP.

$$\frac{A_1:a_1 \ldots A_k:a_k}{A_1 \wedge \ldots \wedge A_k:(a_1,\ldots,a_k)}\ (\wedge+) \qquad \frac{A_1:a_1 \wedge \ldots \wedge A_k:a_k}{A_i:a_i}\ (\wedge-)$$

$$\frac{\begin{array}{c}A:a\\ \vdots\\ B:f\end{array}}{A \supset B:f}\ (\supset+) \qquad \frac{A:a \quad A \supset B:f}{B:f(a)}\ (\supset-)$$

**Figure 4.15.** Inference rules with realizations of formulas

We are not presenting here an *algorithm of structural synthesis of programs*, because it is the same as the algorithm A.4.10 for higher-order functional constraint solving. Indeed, let us return once again to the computability calculus that has been introduced in Chapter 1, and further extended in Section 4.3.4 when we obtained the formula

$$(U_1,\ldots,U_s \to V_1,..,V_t),X_1,\ldots,X_m \to Y_1,\ldots,Y_n$$

that expresses a higher-order functional constraint. Interpreting arrows as implication symbols, letters as propositional variables and commas as conjunction symbols in the formulas of the language for higher-order functional constraints, one gets exactly the formula we have derived as a specification of a program with a functional argument in SSP. Introducing suitable admissible inference rules of intuitionistic propositional logic, one gets precisely the computability calculus that derives the programs for solving problems on constraint networks with higher-order functional constraints.

It has been observed that the logical language we have just described is powerful enough for encoding any intuitionistic propositional theorem. This result gives us an unpleasant complexity estimate: the time complexity of planning a solution of a problem on a higher-order functional constraint network is exponential. Still, there are good strategies of proof-search for various cases in this calculus [32].

## 4.7. Planning

*Planning* is synthesis of an algorithm for achieving a given goal. It can be, in principle, done by means of program synthesis methods. However, there are several specific features of planning that prevent the usage of conventional program synthesizers:

- nonmonotonous character of achieved subgoals – performing a new action may invalidate some achieved subgoal
- *frame problem* – any action may have indirect influences on the situation where action is performed.

Plans tend to be shorter and with simpler structure than computational algorithms. Common formalisms for planning are

- *first-order logic*
- *situation calculus.*

Quite helpful is *hierarchical planning*, when only a coarse-grained plan is built first. One uses often linear and branching plans, iterative and recursive plans are being used seldom. *Naive planning* (that is the most common planing method) uses knowledge about a set of possible actions where, for each action, its pre- and postconditions are given (i.e. knowledge is again represented in the form of production rules).

A common planning example is planning in a block world (see Fig. 4.16), where there are actions for moving blocks:

*PICKUP, PUTDOWN, STACK, UNSTACK.*

An example with pre- and postconditions is the following:

*PICKUP(block)* ***possible when*** *ONTABLE(block)* ***and*** *CLEAR(block)* ***gives*** *HOLDING(block)* ***and*** *NOT CLEAR(bloc)* ***and*** *NOT ONTABLE(block).*

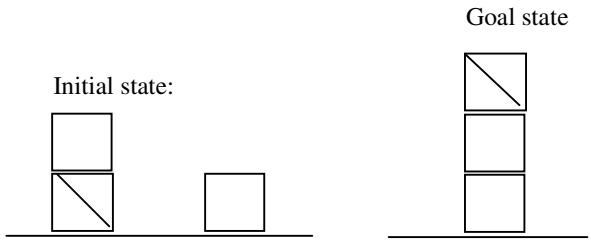



**Figure 4.16.** Plannig problem in a block world

We can see the similarity with a rule-based knowledge representation here. The planning algorithm is similar to the algorithm of application of production rules, see Chapter 1.

*Situation calculus* uses predicate logic extended with situation constants. Each literal, e.g. a literal that states that some object *x* is above *y*:

*Above(x,y)*

is extended with an argument showing the situation, (e.g. time moment usually) when this literal is valid:

*Above(x,y,s1)*

This requires much information and creates the *frame problem*: when proceeding from one situation to another, all literals must be recomputed, or there must be some rules for deciding which literals remain unchanged.

*Hierarchical planning.* To avoid large amount of search, planning can be performed in several stages: first rough planning that produces a plan consisting of *macro-actions* that must be made precise either by more planning or by macro substitution.

### 4.7.1. Scheduling

Planning with time constraints is *scheduling*. In principle, one can specify a scheduling problem by means of a network similar to a functional constraint network. However, adding durations (or some other resources) to functional constraints and asking for an optimal solution, one gets a NP-complete problem that is difficult to solve. Impressive examples of automatic scheduling by means of program synthesis technique exist, e.g. PLANWARE developed in the Kestrel Institute that has been able to solve very large logistic problems of scheduling military air transportation in the Pacific area [45].

## 4.8. Intelligent agents

There is an approach to the AI where any intelligent system is considered as an *intelligent agent* or a collection of them [39]. This leads to a very broad definition of an intelligent agent. We are going to use a more specific concept of intelligent agent. First of all, we can agree that intelligent agents are software components. Besides that, they possess some features of intelligent behavior that makes them special:

- *proactiveness*
- *understanding of an agent communication language* (*ACL*)
- *reactivity* (ability to make some decisions and to act).

Te agents may have more special properties like

- *planning ablity*
- *mobility*
- *perception of environment*
- *reflection*.

In the software engineering community there is a concept of *software agents*, and they are considered to be objects that are at least proactive and have the ability to use some language – an agent communication language - ACL. (Comparing agents and objects, one can say that objects may be passive, and they do not have to understand any language.) *Proactiveness* is the ability to act without external stimuli. It can be realized by means of a separate thread performing the simple algorithm A.4.13, using a knowledge base of proactive actions. Below is a simple *algorithm of proactive behavior*. The main part of the algorithm is hidden in the function *consultKB()* that takes into account the time moment, and chooses from a knowledge base an action or a sequence actions that must be performed.

**A.4.13:**

```
while true do
        wait(givenInterval);
        lookAtClock();
        consultKB();
        act();
od
```

An agent has to act in an open and nondeterministic world. Therefore a good agent should be able to do the following:

- perceive the world
- affect the world
- have a (rational) model of behavior
- have motivations to be fulfilled by implementing corresponding goals.

Even the simplest agents can perceive the world and affect the world by reacting on stimuli coming from the world. It is assumed that perceiving the world and affecting the world may include some ability to communicate by means of an agent communication language with other agents and humans.

## 4.8.1. Agent architectures

The functions listed above must be supported by agent architecture. Now we are going to consider *architectures of agents* beginning from the most simple one – reactive agent shown in Fig. 4.17. This figure shows just the principal blocks needed for the algorithm A.4.13 describing the proactive behaviour. (From here on we assume that proactiveness is always built into the agent, and is not explicitly shown in the architecture.)

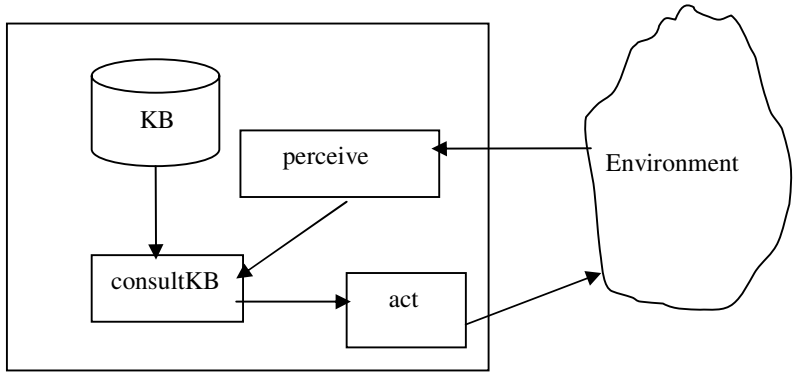



**Figure 4.17**. Simple reactive agent

The *simple reactive agent* architecture is too limited for real agents, and does not support even memory and learning abilities. The next architectural level is *agent with internal state*, shown in Fig. 4.18. It includes explicitly an internal state of the agent. The state can be changed immediately by perceiving of environment, but also by updating it on

the basis of information from a knowledge base. The blocks *state* and *KB* are shown in figure by dashed lines, expressing that these are passive blocks (data blocks). The block *select* consults the *KB* and chooses an action to be performed by the block *act*.

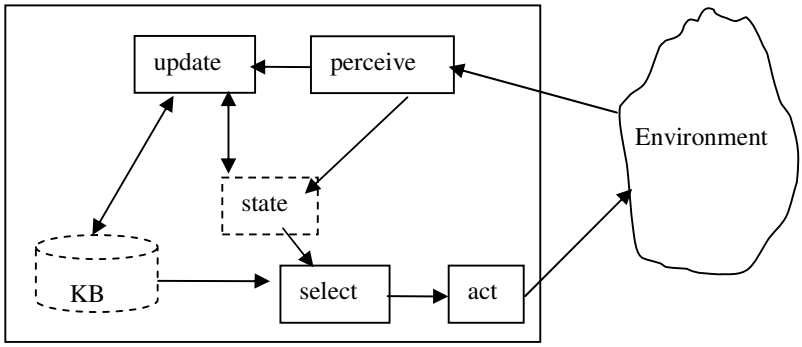



**Figure 4.18.** Agent with internal state

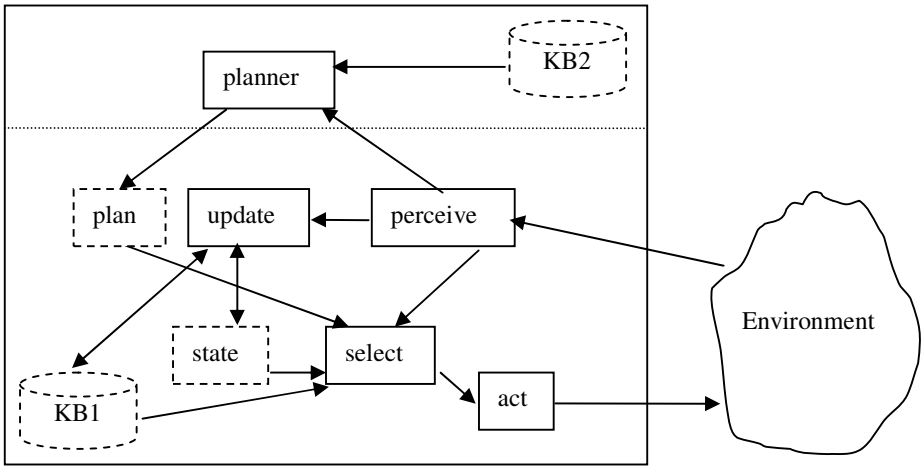



**Figure 4.19**. Goal-driven agent

Real agent architectures include planning that sets goals to acting level. Fig. 4.19 shows architecture of a *planning agent*. In this figure we see a new knowledge base for planning, and a plan that influences the selection of actions. No goals are explicitly shown in Fig. 4.19, but we have to distinguish between a goal for planner and goals for acting. In the algorithm of a goal-driven agent one can see two threads: *planner* and *actor*, the first for planning and the second for acting. The following notations are used in the *goal-driven agent's algorithm*:

*newGoal* – true, if a new goal for planner is detected;
*newPlan()* – the planning procedure;
*perceive()* – the procedure that gets input from the environment and from clock;
*update()* – updates the current state and knowledge base;
*select()* – selects an action;
*act()* – performs the action.

**A.4.14:**

```
planner: while true do
        if empty(plan) or newGoal then
                plan=newPlan();
                wait()
        fi
od

actor: while true do
        perceive();
        update();
        select();
        act();
        wait()
od
```

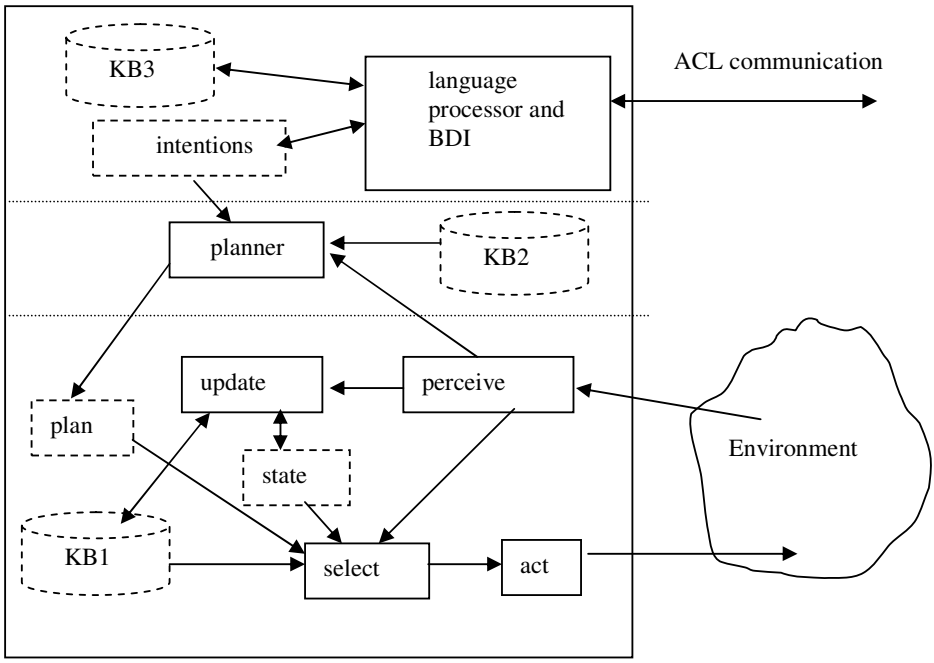



**Figure 4.20.** Agent with beliefs, desires and intentions

Finally, an advanced agent architecture includes a level of beliefs, desires and intentions. This agent architecture is called briefly *BDI.* This architecture, see Fig. 4.20, is closely related to *agent communication language* (ACL). At a closer look, beliefs are knowledge of an agent that it has about the world (agent believes that things are as it knows they should be). Agent's proactivity gives him desires. The desires become a source of intentions: an intention appears as soon as an agent finds how to fulfil the desires. Intentions become goals for the planner on the level of planning, as shown in Fig. 4.20.

*Algorithm of BDI agent* includes three threads: *communicator* for language processing, *planner* for planning and *actor* for acting. It uses a procedure *communicate()* – for communicating with other agents.

**A.4.15:**

```
communicator: while true do
                  communicate();
                  wait()
              od

planner: while true do
              if empty(plan) or newGoal then
                  plan=newPlan();
                  wait()
              fi
         od


actor: while true do
              perceive();
              update();
              select();
              act();
              wait()
         od
```

We have ignored the details of synchronization in the multithreaded algorithms of agents, just showing the *wait()* operators in threads.

## 4.8.2. Agent communication languages

There are several widely accepted agent communication languages (ACL). One of the oldest is Knowledge Query and Manipulation Language – KQML and a more recent is FIPA-ACL. The KQML language is based on the speech act theory:

- it prescribes a message format and message handling protocol
- it has primitives with fixed meaning, called *performatives*.

KQML performatives have the role of commands. KQML performatives are:

| | |
|---|---|
| *ask-if* | *ask-one* |
| *ask-about* | *ask-all* |

| | |
|---|---|
| *reply* | *deny* |
| *tell* | *advertise* |
| *untell* | *cancel* |
| *evaluate* | *achieve* |
| *subscribe* | *register* |
| *unregister* | *monitor* |

It is assumed that the environment for using KQML provides facilitators - special agents supporting the communication process, e.g. supporting a name service etc. A KQML message consists of

- performative
- its parameters, if any
- message content (not understood in KQML)
- communication (transport and context) information.

Example of a message asking the price of a VCR in US dollars:

```
(ask-one                                  - performative
  :sender customer1                       - communication
  :content (PRICE VCR ?price)             - content
  :reply-with VCR-price
  :langugae LPROLOG                       - parameters
  :ontology USD
)
```

In order to be able to communicate, agents must share an ontology that is a system of related concepts. This is a hot topic in web-based software, especially, in web services and semantic web. We have briefly discussed ontologies in Chapter 1.

### 4.8.3. Implementation of agents

Let us look now briefly at the implementation of agent architectures. A good idea is to use *agent shells* – universal agents with almost empty knowledge bases. They have all the needed "machinery" for performing as an agent, but only the most basic universal knowledge, needed for every agent.

One of the first agent shells was proposed by Shoham [44], and called Generic Agent Interpreter. This agent shell is shown in Fig. 4.21, where control flow is depicted by continuous lines and data flow – by dotted lines. As we can see here, the agent starts its "life" from initialisation of its functions. Thereafter it performis in an infinite loop, taking into the account time, and communicating with external world (including other agents).

*Agent middleware* has been developed to facilitate the development of agents and to support their communication. A popular agent development environment is Jade – Java Agent DEvelopment Framework, see *http://sharon.cselt.it/projects/jade/.* JADE is a software framework fully implemented in Java language. It simplifies the implementation of multi-agent systems through a middleware that claims to comply with the FIPA specifications (*http://www.fipa.org/*) and through a set of tools that support the debugging and deployment phase. The agent platform can be distributed across computers with different operating systems, and the configuration can be controlled via a remote GUI. The

configuration can be even changed at run-time by moving agents from one machine to another, as and when required.

Another agent tool is Agora. It is an infrastructure for supporting cooperative work in *multi-agent systems* (MAS). The infrastructure is based on a concept of a facilitator of cooperative work. An example of a Virtual Shopping Mall as a general framework for modeling agent-based intelligent software services in mobile communications is used for illustration of the approach [31].

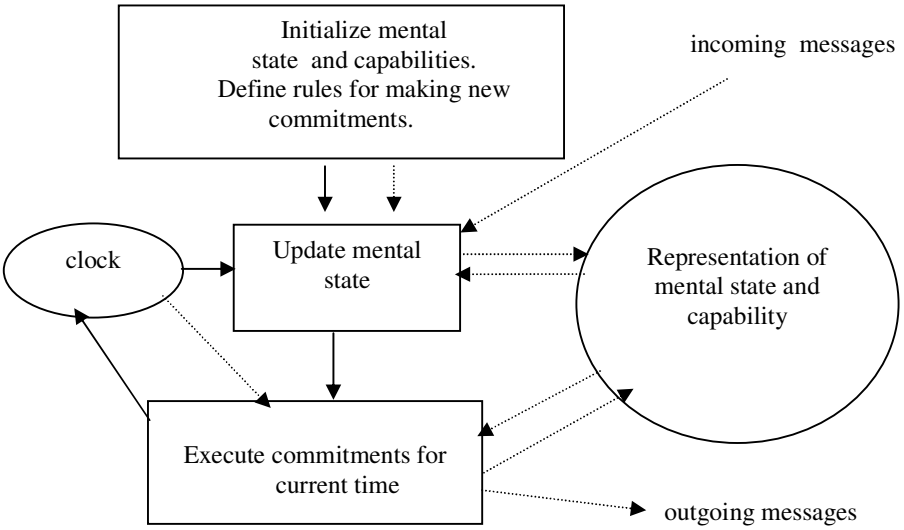



**Figure 4.21**. Generic agent interpreter

## 4.8.4. Reflection

An advanced feature of an agent is *reflection*. This is an ability to perceive and take into account agent´s own state. In this case an agent has to monitor itself, and to plan its own actions on the basis of this information. The simplest is *procedural reflection*. In this case an agent has some preprogrammed procedures that are switched on when a state of the agent requires so.

A more advanced way to implement agent's reflection is to keep a model of agent's self and to use it for planning agent´s actions. This is called *declarative reflection* [1]. Fig. 4.22 shows an implementation scheme of declarative reflection. In this case an agent operates according to plans that it constructs in order to achieve given goals on the joint model of the self and the external world. This model is described in a high level modeling language, because it must be updated every time when the agent's state or the external world change. This language can be the language of computational frames presented in Chapter 1, see also [1]. The model is described and updated by a *reflection manager*. Detecting changes in the self requires some kind of internal observers. These observers are implemented as permanently running threads – *daemons*. Signals from external world can be either handled by daemons or directly passed to the reflection manager, but changes in the state of the agent self can be observed only by daemons.

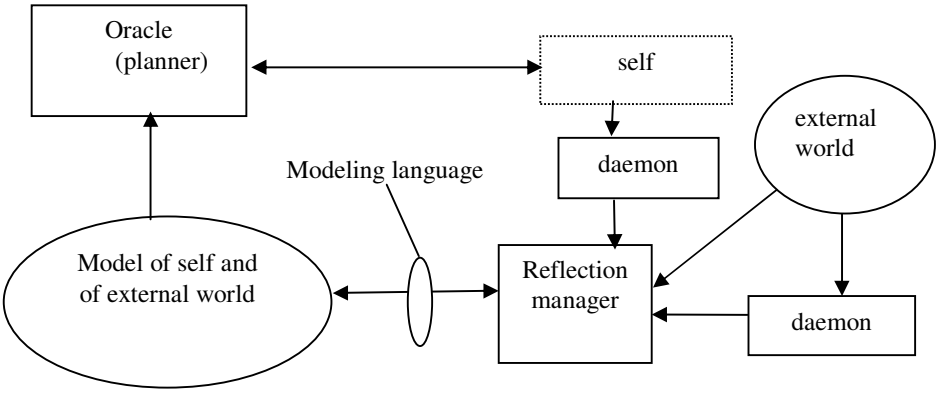



**Figure 4.22**. Declarative reflection in an agent

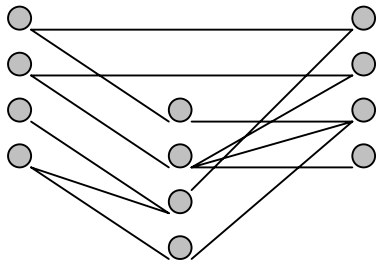
Declarative reflection is more flexible and powerful than procedural reflection, because it includes planning of control of agent's actions on a high-level conceptual model, not only switching on and off low-level preprogrammed procedures.

## 4.9. Exercises

1. Write a computational frame of an electric circuit of two resistors *r1* and *r2* connected in parallel using the Ohm's law *u=i*r* for both resistors *r1* and *r2*. Draw a constraint net of the computational frame.

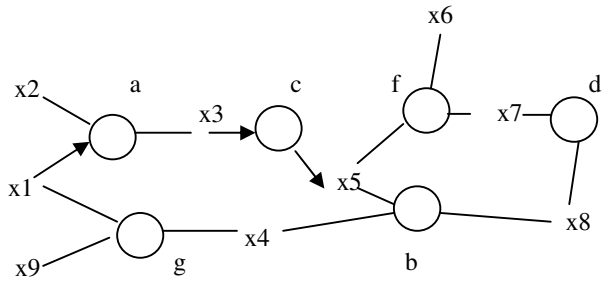
2. Make the binary constraint *R={(1,3),(1,4),(2,3),(3,3),(2,2)}* arc consistent, if its domain and range are both *{1,2,3,4}*.

3. Make the constraint network shown below path consistent. The network has three variables and three binary constraints. The small circles are possible values of variables. Each line binds a pair of values belonging to the respective relation (to the constraint).



4. Decide about solvability of the following problems on FCN shown below:

x1,x3→*x5*
*x1,x3,x9→x6*
*x1,x9→x5*



5. Find maximal solvable problems on the FCN form above.

6. Given functional (and higher-order functional) constraints

$f: q \rightarrow b,c$
$g: (u \rightarrow y),b->z$
$h: c,u \rightarrow d,y$
$k: d \rightarrow i$

a) write out higher-order functional constraint(s)

b) draw the constraint network constituted by the constraints

c) show, which of the following problems on the constraint network are solvable:

$q \rightarrow i$
$q \rightarrow z$
$c,u \rightarrow i$

d) for each solvable problem write the sequence of functional dependencies that solves it.

7. Calculate interval of values for $c$ on the interval constraint network that has the following constraints and given intervals of values:

$a=b+c$
$f=b/q$
$q*2=k$

$a \in [100.0,100,1]$
$f \in [10.11,10.21]$
$k \in [5.5,5.6].$

8. Derive the formula $\supset b$ in the intuitionistic propositional calculus from the axioms

$(y \supset a) \supset b$
$(a \supset b) \supset x$
$x \wedge y \supset a$

and, if needed, then from axiom schema $F \supset F,$ where $F$ is any propositional variable.

Do it, first using the conventional inference rules, and then using the SSP rules. Compare the complexity of proof search in these cases.

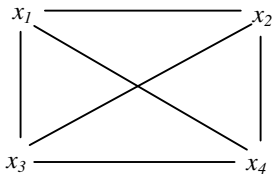
9. Draw a higher-order constraint network that represents the following double integral:

$$s=\int_0^1 \int_0^{g(y)} u(x,y)dxdy$$

Hint -- use an intermediate variable *w* for the value of the internal integral:

$$w=\int_0^{g(y)} u(x,y)dx.$$

10. How many different elements at least must be in the domains of variables $x_1$, $x_2$, $x_3$, $x_4$ of the following binary constraint network in order to have a solution for the constraints that all are "not equal"?




11. Develop an algorithm of calculation of bounds *la, ua* for *a=b/c*, where

*lb<0, ub<0*
*lc<0, uc<0.*

This page intentionally left blank

# References


1. Addibpour, M., Tyugu, E. Declarative reflection tools for agent shells. *Future Generation Computer Systems*. July 1996, 1 - 12.

2. Adelson-Velski, G., Arlazarov, V., Donskoi, M. Some methods of controlling the tree search in chess programs. *Artificial Intelligence*, v. 6, No. 4, 1975, 361 – 371.

3. Bellman, R. *Dynamic Programming*. Dover Publications, 2003.

4. Biermann, W., Kirshnaswany, R. Constructing programs from example computations. *IEEE Trans. On Software Engineering,* vol. 2, Sept.1976, 141 –153

5. Bitner, J., Reingold E. Backtrack Program Techniques. *CACM*, v.18(11) 1975, 651-656.

6. Bratko, I. Prolog *Programming for Artificial Intelligence*. Third edition. Pearson Education, Addison-Wesley, 2001.

7. Burstall, R. M., Darlington, J. A transformation system for developing recursive programs. *J. ACM,* 24(1), 1977, 44-67

8. Cutland, N. Computability, *An introduction to recursive function theory*. Cambridge University Press, 1980.

9. Damas, L., Moreira, N., Broda, S. Resolution of Constraints in Algebras of Rational Trees. *EPIA 1993*. LNCS, v. 727, Springer, 1993, 61-76

10. Freeman-Benson, B., Maloney, J., Borning, A. An Incremental Constraint Solver. *CACM*, v. 33, No.1, 1990, 54 - 63.

11. Genesereth, M., Nilsson, N. *Logical Foundations of Artificial Intelligence.* Morgan Kauffmann, 1986.

12. Gergel, V., Strongin, R.: Parallel computing for globally optimal decision making on cluster systems. *Future Generation Comp. Syst. v. 21*(4) 2005, 673-678.

13. Goldberg, D. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley, 1988.

14. Gold, E. Language Identification in the Limit. *Information and Control*, v. 10, 1967, 447-474.

15. Green, C: Application of Theorem Proving to Problem Solving. *IJCAI 1969*. 1969, 219-240

16. Grigorenko, P., Saabas, A., Tyugu, E. Visual Tool for Generative Programming. *Proc. of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13).* ACM Publ. 2005, 249 – 252.

17. Hart, P., Nilsson,N., Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Trans. Sept. Sci. Cybern*. v. SSC-4, (2), 1968, 100-107.

18. Holland, J. *Adaptation in Natural and Aritificial Systems*. Univ. of Michigan Press, 1975.

19. Ibaraki. T. Theoretical Comparison of Search Strategies in Branch and Bound. *Int. Journal of Comp. Inf. Sc.* , 1976, 315 - 344.

20. Karzanov, A., Faradjev, I. Solution Planning for Problem Solving on Computational Models. *System Programming and Computer Software.* No.4 1975, (Russian).

21. Kohonen, T. *Self-Organization and Associative Memory*. Springer, 1984.

22. Kumar, V.,Kanal, L. A general branch and bound understanding and synthesizing and/or tree search procedures. *AI*, v.21 (1), 1983, 179-198.

23. Lavrov, S., Goncharova L. *Automatic Data Process*ing. Nauka, Moscow, 1971 (Russian).

24. Lippmann, R. An Introduction to Computing with Neural Nets. *IEEE ASSP Magazine*, No. 4, 1987, 4 – 20.

25. Lorents, P. Formalization of data and knowledge based on the fundamental notation-denotation relation. *IC-AI´2001 International Conference. Proceedings*. v. 3. CSREA Press, 2001, 1297 – 1301.

26. Lorents, P. A System Mining Framework. *Proc. of The 6th World Multiconference on Systemics, Cybernetics and Informatics.* v. 1, 2002, 195 – 200.

27. Mackworth, A. Constraint Networks. In: Shapiro, S. (Ed.) *Encyclopedia of Artificial Intelligence*, v. A, B. Wiley & Sons, 1992, 276 - 293.

28. Manna, Z., Waldinger, R. Towards automatic program synthesis. *Comm.ACM, 14(3), 1971*, 151 –164

29. Maslov S. *Theory of Deductive Systems and its applications*. The MIT Press, 1987.

30. Maslov, S. Maslov S. An inverse method of establishing deducibilities in the classical predicate calculus, *Soviet Mathematics, Doklady*, 1964, 1420-1423.

31. Matskin, M., Kirkeluten, O., Krossnes, S., Sæle, Ø. Agora: An Infrastructure for Cooperative Work Support in Multi-Agent Systems. *Autonomous Agents 2000: Workshop on Infrastructure for Scalable Multi-agent Systems*, Barcelona, June 3-7, 2000, 22-28.

32. Matskin, M., Tyugu. E. Strategies of Structural Synthesis of Programs and Its Extensions. *Computing and Informatics*. v.20, 2001, 1 -25.

33. Michalski, R., Carbonell, J., Mitchell, T. (Eds.) *Machine Learning: An Artificial Intelligence Approach.* Tioga, Palo Alto, 1983.

34. Michie, D. The state of the art in machine learning. In: *Introductory readings in expert systems.* Gordon and Breach. New York, 1982, 209 – 229.

35. Minsky, M. *A Framework for Representing Knowledge*. MIT-AI Laboratory Memo 306, June, 1974.

36. Mints, G., Tyugu, E. Justification of the Structural Synthesis of Programs. *Science of Computer Programing,* v. 2, No. 3, 1982, 215 - 240.

37. Muggleton, S., Buntine, W. Machine invention of First-Order Predicates by Inverting Resolution. *Proc. 5th International Conference on Machine Learning*. Morgan Kauffmann, 1988.

38. Mullat, J. E., A fast algorithm for finding matching responses in a survey data table. Mathematical Social Sciences, Elsevier, vol. 30(2), 1995, 195-205.

39. Norvig, P., Russell, S. *Artificial Intelligence: Modern Approach*. Prentice Hall, 2000.

40. Pearl J. *Intelligent Search Strategies for Computer Problem Solving*. Addison Wesley, 1984.

41. Quinlan, R. "Learning Logical Definitions from Relations", Machine Learning 5, 1990, 239-266.

42. Quinlan, R. Induction of Decision Trees. *Machine Learning* v.2, Kluwer Publ., 1986, 81-106.

43. Robinson, J. A. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM,* v. 12, 1965, 23 - 41.

44. Shoham, Y. An Overview of Agent-Oriented Programming. In: Bradshaw, J. M. ed. *Software Agents*. AAAI Press/The MIT Press, 1997, 271 - 290.

45. Smith, D. and Parra, E. Transformational Approach to Transportation Scheduling. *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, IEEE Computer Society Press, 1993, 60-68.

46. Shapiro, S. C. *Encyclopedia of Artificial Intelligence*. Wiley, New York, 1992.

47. Statman, R. Intuitionistic propositional logic is polynomial-space complete. *Theoretical Computer Science* 9, 1979, 67–72.

48. Tyugu, E. Understanding Knowledge Architectures. *Knowledge-Based Systems*. Vol. 19, No. 1, 2006, 50–56..

49. Viterbi, A. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory* 13(2) 1967, 260–269.

50. Vyhandu, L. Fast Methods in Exploratory Data Analysis. *Proc. Ta*

51. Tyugu, E. *Knowledge Based Programming*. Addison Wesley, 1988*llinn Techn. Univ*., v. 705, 1989, 3 – 13.

52. Zadeh, L. A fuzzy-algorithmic approach to the definition of complex or imprecise concepts, *Int. Jour. Man-Machine Studies* 8, 1976, 249-291.

# Subject index

This page intentionally left blank

This page intentionally left blank

This page intentionally left blank